Multi-Path Planning for Hydraulic Fluid Routing

by

Landon Carter

S.B., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Dr. David R. Wallace
Professor of Mechanical Engineering; MacVicar Faculty Fellow
Thesis Supervisor

Accepted by .....
Dr. Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

Multi-Path Planning for Hydraulic Fluid Routing

by

Landon Carter

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed, implemented, and optimized an algorithm to solve the circuit-routing problem, optimizing the solution for exact correctness in a low aspect ratio scenario, as opposed to approximate correctness in high aspect ratio scenarios, where topological approaches are typically applied. I applied this algorithm to 3D printed hydraulically actuated robots, though it has additional applications in circuit routing for PCB assembly, FPGA interconnect optimization, fiber optic routing, and other routing applications. The performance of the algorithm is discussed, profiled, and tuned from an algorithmic perspective, with further improvements suggested. The effect of starting conditions on the performance of the algorithm is discussed theoretically and analyzed in real-world performance. Overall, the algorithm is shown to provide exactly correct results and perform adequately over a range of starting conditions useful for 3D printed hydraulic fluid pipes.

Thesis Supervisor: Dr. David R. Wallace
Title: Professor of Mechanical Engineering; MacVicar Faculty Fellow

Acknowledgments

I would like to thank Dr. David Wallace for a true learning experience this year in the subtle balance of perfection and completion, as well as the freedom to explore creatively and contribute to 2.009 and 2.744. I would also like to thank my fellow TA's throughout the semesters. From 2.009: Grace Li, the perfect-pixel-pusher, Chuck Xia, the animated animator, and Georgia Van de Zande, the rockin' roller. From 2.744: Audrey Bosquet, the illuminating illustrator, and Josh Ramos, the persevering PDL-er. To the rest of cadlab - Victor, Dabin, and Geoff, especially - thank you.

I would like to extend further thanks to those who supported the origin of this research - Dr. Robert MacCurdy and Dr. Daniela Rus, from the Distributed Robotics Lab, thank you for endless patience and guidance in developing the project. Thank you additionally to Jeff Lipton, Cenk Baykal, and Alexander Wallar for productive discussions in the early course of algorithm development.

Thank you to my parents, family, and friends. Through thick and thin, I have always had a solid foundation of support, for which I am eternally grateful.

And finally, thank you to the institute for 5 years of exploration, friend-making, robot-building, quadcopter-flying, picture-taking, PSETs, and projects. Sometimes, all at once, and certainly without a wink of sleep. IHTFP.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The circuit routing problem is: given n pairs of coordinates $(x_{1,i}, y_{1,i}, z_{1,i}), (x_{2,i}, y_{2,i}, z_{2,i})$, $i \in 1 \dots n$, on a 3D grid with dimensions $x \times y \times z$, find the set of shortest paths $(p_1, p_2, \dots p_n)$ of points connecting these coordinates such that for all vertices, if $i \neq j$, $v \in p_i$ then $v \notin p_j$ (eg, the paths do not intersect). Here, the set of shortest paths is the set which has the shortest total length (where length is the sum of pairwise Euclidean distances of consecutive elements).

This problem has numerous real-world examples, such as its namesake: circuit routing for PCB assembly, as well as FPGA interconnect optimization and 3D IC design optimization. Other applications include the application which inspired this work - routing hydraulic fluid pipes throughout 3D-printed hydraulically actuated robots, as well as other large-scale physical applications such as routing fiber optics, underground tunnels, and similar.

Chapter two describes the Multi-Path A* (MPA*) algorithm and proves its correctness in exactly solving the circuit-routing problem.

Chapter three discusses the application of MPA* to hydraulically actuated 3D printed robots.

Chapter four discusses the theoretical and measured performance of MPA*, as well as both implemented and future optimizations. Code profiling tools and benchmarking are employed to direct optimization efforts.

Chapter five summarizes the results and proposes opportunities for future work.

1.1 Related Work

There are many algorithms [2, 3, 4, 5, 6, 7] which approximate the circuit-routing problem, especially for cases where $x, y \gg z$, which is the case most frequently seen for 2.5D applications, such as routing traces in a PCB. The vast majority of these algorithms focus on the case where $z = 2$, and in particular where the top layer contains purely x-oriented wires and the bottom layer contains purely y-oriented wires. Other variants include the via minimization problem in the constrained (topology predetermined), or unconstrained (topology optimized at runtime) flavors [8]. One particular variant of the 2-layer version called Dogleg Channel Routing has been proven to be NP-Complete [9], and it would be unsurprising to find that most variants of the problem, including the full-3D circuit routing problem, are NP-complete, but further explorations into this are beyond the scope of this thesis. More modern extensions of these algorithms include photonic waveguide routing [10], as well as many heuristic approximate solutions for professional PCB design software, FPGA design software, and 3D IC design software.

For 3D-printed robots with hydraulic fluid channels, the assumption that $x, y \gg z$ is not valid, and the 2-layer fixed-orientation algorithms as well as via minimization algorithms all perform inadequately for routing these hydraulic fluid channels. Such robots are described by MacCurdy et al [11]. Solving this problem where $x, y \not\gg z$ also has applications in 3D-printed electronics and optics [12, 13].

Chapter 2

Multi-Path A*

2.1 Algorithm Overview

MPA* has been implemented in `Python`, focusing on memory efficiency by implementing data structures lazily when possible. This includes a re-implementation of the lazy optimizations introduced by Koenig et al in Lifelong Planning A* [1]. The overall graph search section's implementation is also lazy, generating children nodes from parent nodes only as needed.

At a high level, MPA* evaluates each path as an independent Lifelong Planning A* (LPA*) search [1], adding constraints (by marking edges as impassable) in a branching fashion for replanning steps, while also maintaining a priority queue of which set of constraints is most likely to lead to an optimal solution, and evaluating in order to search the constraint space efficiently. Using LPA* for each path allows fast replanning when a constraint is added.

MPA* begins in `InitializeMPA()` by initializing an LPA* search for each of the paths. After initializing these, MPA* inserts the list of LPA* searches into a priority queue, U, which orders LPA* lists to evaluate based on the minimum cost they could achieve.

During the evaluation phase, `ComputeShortestPaths()`, MPA* calls

`ProcessLPAList()`, which computes the paths for the LPA* list at the top of the priority queue, then determines the cost of the set of paths generated. Each execution of this beyond the first one constitutes a "replanning" step. For each pair of paths, MPA* calls `CheckOrBranch()`, which determines if the two paths intersect or come within a minimum distance (the diameter of our pipes). If the two paths intersect, the edges at fault for each path are identified. In each branch direction, an edge is marked impassable for the corresponding LPA*, and then the modified LPA* list is inserted into the priority queue with a cost equal to the cost of the parent LPA* list that has just been evaluated.

One important note for the correctness of MPA* is that adding a constraint to an existing LPA* will not decrease its cost - this is a guarantee provided by LPA*. Therefore, adding a constraint to one LPA* out of a list of LPA* searches will not decrease the cost of the set of paths generated. Thus, once we find a set of constraints which produces non-intersecting paths, we have an upper bound on the minimum possible cost of routing the paths.

After finding a set of constraints which produces a valid set of paths, MPA* updates *minCost* and *best* in line 7 of `ComputeShortestPaths()`. As discussed, this sets an upper bound on the minimum cost, so in order to find the absolute minimum cost, MPA* simply needs to evaluate each remaining LPA* list in the priority queue which could improve the upper bound. If no valid path is ever found, the priority queue will empty after exhaustively searching the entire constraint space, and MPA* will return *None*, indicating that there is no set of paths which is nonintersecting.

```
   procedure InitializeMPA(desiredPaths):
1      initialLPAlist = [∅]
2      for (start, goal) ∈ desiredPaths:
3          initialLPAlist.append(InitializeLPA(start, goal))
4      U.Insert(initialLPAlist, 0)
   procedure ComputeShortestPaths():
1      minCost = ∞
2      best = None
3      while U.TopKey() < minCost:
4          LPAlist = U.Pop()
5          (tmpPaths, cost) = ProcessLPAList(LPAlist)
6          if tmpPaths ≠ None:
7              update minCost, update best
8      return best
   procedure ProcessLPAList(LPAlist):
1      paths = [ ]
2      for LPA ∈ LPAlist:
3          LPA.ComputeShortestPath()
4          paths.append(LPA.GetPath())
5      c = Σ_{p∈paths} PathCost(p)
6      for i ∈ range(length(paths)):
7          for j ≠ i ∈ range(length(paths)):
8              if !CheckOrBranch(LPAlist, paths, i, j, c):
9                  return (None, ∞)
10     return (paths, c)
   procedure CheckOrBranch(LPAlist, paths, i, j, c):
1      if e_1 ∈ paths[i] intersects with e_2 ∈ paths[j]:
2          split_1 = LPAlist[i].MakeEdgeImpassable(e_1)
3          split_2 = LPAlist[j].MakeEdgeImpassable(e_2)
4          U.Insert(split_1, c)
5          U.Insert(split_2, c)
6          return False
7      return True
```

Algorithm 1: Multi-Path A* (MPA*).

2.1.1 LPA*

Lifelong Planning A* (LPA*) [1] is an incremental version of A* that allows for efficient path replanning when edge weights are updated. See Algorithm 2 for an overview.

LPA* maintains two estimates of the goal distance: $g(s)$ and $rhs(s)$. g is updated in the same way it would normally be for A*, and rhs satisfies the following invariant

$$rhs(s) = \begin{cases} 0 & s = s_{goal} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases}$$

If $g(s) = rhs(s)$, s is locally consistent, and is locally inconsistent otherwise. U is a priority queue that LPA* maintains with locally-inconsistent states. The keys for the priority queue are $k = [\min(g(s), rhs(s)) + h(s), \min(g(s), rhs(s))]$, and are sorted lexicographically. LPA* proceeds by popping states off of the priority queue and recalculates its g-value - if the state is overconsistent ($g(s) > rhs(s)$), it sets $g(s) = rhs(s)$, and if the state is underconsistent, it sets $g(s) = \infty$ and calls `UpdateVertex(`s`)` to propagate and resolve any potential changes. LPA* stops when s_{start} is locally consistent.

procedure `CalculateKey(`s`)`**:**
1 **return** $[\min(g(s), rhs(s)) + h(s, s_{goal}); \min(g(s), rhs(s))]$
procedure `InitializeLPA()`**:**
1 $U = \emptyset$
2 $\forall s \in S : rhs(s) = g(s) = \infty$
3 $rhs(s_{start} = 0$
4 $U.Insert(s_{start},$ `CalculateKey(`s_{start}`)`
procedure `UpdateVertex(`u`)`**:**
1 **if** $u \neq s_{start}$**:**
2 $rhs(u) = \min_{s' \in Pred(u)}(g(s') + c(s', u))$
3 **if** $u \in U$**:**
4 $U.Remove(u)$
5 **if** $g(u) \neq rhs(u)$**:**
6 $U.Insert(u,$ `CalculateKey(`u`)`$)$
procedure `ComputeShortestPath()`**:**
1 **while** $U.TopKey() <$ `CalculateKey(`s_{goal}`)` $||$ $rhs(s_{goal} \neq g(s_{goal}))$**:**
2 $u = U.Pop()$
3 **if** $g(u) > rhs(u)$**:**
4 $g(u) = rhs(u)$
5 $\forall s \in Succ(u)$: `UpdateVertex(`s`)`
else:
6 $g(u) = \infty$
7 $\forall s \in Succ(u) \cup \{u\}$: `UpdateVertex(`s`)`

Algorithm 2: Lifelong Planning A* (LPA*) [1].

2.1.2 Priority Queue Considerations

There are two places where a priority queue is utilized in MPA*: the overall priority queue for `LPAlist`'s, U, and within the LPA* subroutine for storing nodes.

In the first instance, the queue needs to support `insert`, `pop`, and `topKey`. This is simple to implement with good performance - a min heap will provide $O(\log n)$ `insert` and `pop` and $O(1)$ `topKey`.

In the second instance, the queue needs to support `insert`, `pop`, `top`, `contains`, and `remove`. This is much more complicated to implement with good performance. A min heap, even augmented with a hashtable, will have $O(n)$ performance on remove. Because of LPA*'s particular usage pattern, `insert`, `pop`, and `remove` are called a nearly identical number of times, so their performance must be balanced. Table 2.1 provides a comparison of performance characteristics of different data structures. Note that although Fibonacci heaps typically lead to superior priority queue performance, it does not support `remove` without significant modification and extension.

Table 2.1: Comparison of data structure performance for use in LPA* priority queue

Data Structure	`insert`	`pop`	`contains`	`remove`
binomial heap	$O(1)$	$O(\log n)$	$O(n)$	$O(n)$
binomial heap + hashtable	$O(1)$	$O(\log n)$	$O(1)$	$O(n)$
array	$O(1)$	$O(n \log(n))$	$O(n)$	$O(n)$
array + hashtable	$O(1)$	$O(n \log(n))$	$O(1)$	$O(n)$
linked list	$O(n)$	$O(1)$	$O(n)$	$O(n)$
linked list + hashtable	$O(n)$	$O(1)$	$O(1)$	$O(1)$
red-black tree	$O(\log n)$	$O(\log n)$	$O(\log(n)$	$O(\log n)$
red-black tree + hashtable	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
treap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
treap + hashtable	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
nested lazy-length lists	$O(n^{1/3})$	$O(1)$	$O(n^{1/3})$	$O(n^{1/3})$
nested lazy-length lists + hashtable	$O(n^{1/3})$	$O(1)$	$O(1)$	$O(n^{1/3})$

2.1.3 Caching

In the procedure `CheckOrBranch()`, MPA* must make two copies of the `LPAlist` to be reinserted into U. In practice, this can be a tricky operation - one simple solution is just to perform a deepcopy of `LPAlist`. However, this is very slow. Furthermore, in different branches of MPA*, the same LPA may be produced convergently. Therefore, it would be nice to provide a global cache of LPA's. Therefore, a global cache has been added to MPA*, with keys of the set of impassable edges and nodes. The small amount of overhead to compute a key for each LPA is more than outweighed by the speedup of reduced computational redundancy.

It is also possible for identical `LPAlist`s to be convergently generated - suppose that LPA_1, LPA_2, and LPA_3 all contend for a single node. In one split, LPA_1 and LPA_2 will independently be denied access to the node. In further splits of each of those branches, LPA_2/LPA_3 and LPA_1/LPA_3 will be split to form 4 total branches. However, two of them will have LPA_1 and LPA_2 with blocked access to the node. While we could introduce one more layer of caching, on `LPAlist`'s, the chances of

convergent `LPAlists` is small, and the lower caching level will prevent most of the redundant effort. The primary effect will be that U will grow unnecessarily, but even this does not place a significant strain on memory resources, since the LPAs within the redundant `LPAlists` will correctly be shared without duplication.

2.2 Algorithm Correctness

To argue the correctness of MPA*, it is sufficient to argue that MPA* will exhaustively evaluate all relevant sets of constraints, and to note that LPA* guarantees the optimality of the paths generated given a set of constraints, as proved in Koenig's original paper [1]. If MPA* evaluates all relevant sets of constraints, it must find the best one.

In particular, note that the optimal path can be generated by any set of constraints which includes or implies a minimum set of constraints. Essentially, it is possible to add additional constraints which do not affect the solution LPA* returns. Therefore, the optimal solution can be achieved by adding arbitrary constraints to the unconstrained state, so long as the constraints do not affect the optimal solution.

During our branching phase, we create two new branches, each with one additional constraint. Importantly, in *any* non-intersecting solution (not just the optimal solution), at least one of these constraints can be present, because any set of paths which violates both constraints will be intersecting. Therefore, no branching operation will eliminate the optimal solution. Because a branch terminates only when a valid solution is found, and because that valid solution is the optimal solution given that set of constraints, MPA* guarantees that the optimal solution for that branch will be found. And finally, since the root branch begins with no constraints, it does not eliminate anything, so all states are children of the root branch. Therefore, the globally optimal solution is part of the root branch, which implies that MPA* will find the globally optimal solution.

2.3 Algorithm Implementation

All aspects of the algorithm were implemented in `Python`. Aside from the underlying data structures of certain priority queue implementations, and the distance-between-edges function, all pieces were newly implemented for this project. Of note, this is the first `Python` implementation of LPA* that we are aware of, as well as the first n-D implementation (or indeed, the first non-2D implementation) that we are aware of.

For a full code listing, please see Appendix A.

2.3.1 LPA*

LPA* is implemented as class `LPA`, with class-wide g and rhs dictionaries. Nodes are created lazily via the `StateFactoryInterface` interface. The public methods for `LPA` and `StateFactory` are described in tables 2.2 and 2.3 respectively. Within `LPA`,

nodes are stored as `States`, with only a single public method: `pred`, which returns the neighbors (predecessors) of the state. Internally, it stores its own position and LPA* key, and implements comparability methods so that they can be sorted by key (tiebreaking by position, an important feature for red-black trees or treaps), and equality checked by position. See Appendix A.3 and auxiliary file in Appendix A.4 for the implemented code.

Table 2.2: Public methods for `LPA`

Method	Description
`computeShortestPath()`	Corrects over- and under-constrained nodes according to LPA* rules.
`getShortestPath()`	Calculates and returns the shortest path based on the current node costs.
`makeNodeImpassable(pos)`	Sets a cost of ∞ for all edges leading to `pos`.
`makeEdgeImpassable(edge)`	Sets a cost of ∞ to `edge`.
`getConstraints()`	Returns a tuple of `(impassable_nodes, impassable_edges)`.

Table 2.3: Public methods for `StateFactoryInterface`

Method	Description
`makeOrGetStateByPos(pos)`	Checks an internal hashtable for the presence of a node at `pos`, and creates one otherwise. Returns the relevant state.
`updateState(state)`	Sets the internal hashtable of `state` to point to `state`. This is useful for maintaining consistency when g and rhs are updated.

In order to instantiate an LPA*, a child class of `LPA` must be implemented (as well as a `State` child class and `StateFactory` child class), since the default class does not have `makeNodeImpassable(pos)` or `makeEdgeImpassable(edge)` implemented. `State` does not have `pred` implemented, and `StateFactory` is also just an interface. These child classes are implemented as `FluidLPA`, `StateFactory`, ande `NodeState`, respectively. Most of these implementations is very straightforward, and the code can be found in Appendix A.2. Special care and testing were taken to ensure that `FluidLPA` instances could be deepcopy'd correctly and efficiently, and to ensure that `getConstraints` would work with the caching layer of MPA*.

2.3.2 Priority Queue

As discussed in Section 2.1.2, the priority queues used in LPA* heavily affect the performance. Multiple priority queues were implemented and benchmarked, the full performance results of which can be seen in Section 4.4.1. In order to facilitate

benchmarking, a common interface was defined: `Queue`. The public methods are described in table 2.4.

Table 2.4: Public methods for `Queue`

Method	Description
`insert(item)`	Inserts `item` into the underlying data structure.
`pop()`	Removes and returns the top `item`.
`remove(item)`	Searches for and removes `item` from the underlying data structure.
`top()`	Returns the top `item` without removing it.
`topKey()`	Returns the key of the top `item` without removing it.
`__contains__(item)`	Returns whether is in the data structure.
`__len__()`	Returns the number of `item`s in the data structure. Used only for debugging, benchmarking, and printing progress updates.

The data structures implemented were:

- Binomial heap augmented with hashtable, using `Python`'s `heapq` and `set`. This was the first version implemented, and was used for initial benchmarking. This had $O(1)$ `insert`, $O(\log n)$ `pop`, and $O(n)$ `remove`. See Appendix A.5.1.
- Linked list augmented with hashtable, with elements of the hashtable pointing to elements of the linked list. This allowed for $O(1)$ `pop` and `remove`, but $O(n)$ `insert`. See Appendix A.5.2.
- Array augmented with hashtable. Though this only had $O(n \log n)$ pop and $O(n)$ remove, with $O(1)$ `insert` and with excellent amortized performance from `Python`'s built-in `list` and `sort`, this still performed reasonably well. See Appendix A.5.3.
- Treap augmented with hashtable, using `treap` [14]. This was the first high-performance data structure used, with $O(\log n)$ performance on all relevant methods. However, `Cython` extensions proved less portable than pure `Python`. See Appendix A.5.4.
- Nested lazy-length lists augmented with hashtable, using `sortedcontainers` [15]. This is a particularly interesting one. Although asymptotically slower than a treap or Red-Black tree, the nested lazy-length lists take advantage of `Python`'s extremely well-optimized list performance in the same manner that the array augmented with a hashtable did, and performed better in practice, and had a (constant factor) smaller memory footprint. With certain starting conditions, it is expected that Red-Black trees augmented with a hashtable would be optimal. See Appendix A.5.5.

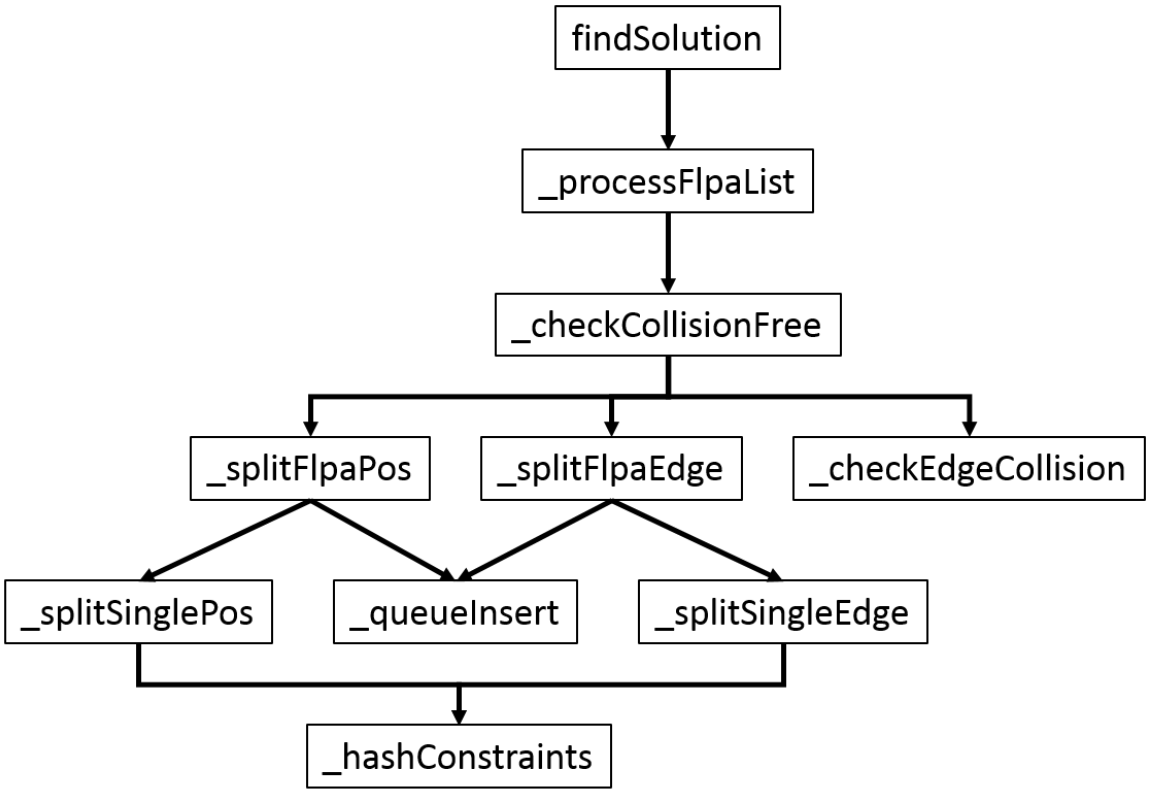


Figure 2-1: Dependency graph of MPA* private methods.

2.3.3 MPA*

Finally, MPA* was implemented. The only public methods are `__init__(desired_routes, valid_region_func)` and `findSolution()`. `__init__` takes a list of tuples of $(start, goal)$ positions and a function which evaluates positions to determine if they are within the defined grid boundary. The private methods are described in Table 2.5, and their dependency graph is shown in Figure 2-1.

Table 2.5: Private methods for `MPA*`

Method	Variables	Description
`_processFlpaList`	`flpa_list`	For each `FluidLPA` in `flpa_list`, execute LPA*. Call the collision-checking method on each pair of `FluidLPA`s.
`_checkCollisionFree`	`flpa_list`, `flpa_index_1`, `flpa_index_2`, `path_1`, `path_2`, `flpa_cost`	Check whether `path_1` and `path_2` intersect, and if they do, split and reinsert into the priority queue.
`_checkEdgeCollision`	`edge_1`, `edge_2`	Checks whether `edge_1` and `edge_2` come within a specified range. The range can be set to allow or disallow certain configurations based on the tube diameter versus the grid size.
`_splitFlpaPos`	`flpa_list`, `flpa_index_1`, `flpa_index_2`, `bad_node`, `flpa_cost`	Splits the specified `FluidLPA`s in `flpa_list` at `bad_node`.
`_splitFlpaEdge`	`flpa_list`, `flpa_index_1`, `flpa_index_2`, `edge_1`, `edge_2`, `flpa_cost`	Splits the specified `FluidLPA`s in `flpa_list` at `edge_1` and `edge_2`, respectively.
`_splitSinglePos`	`flpa_list`, `flpa_index`, `bad_node`	Adds `bad_node` as an impassable node for the specified `FluidLPA`, and looks up the resulting set of constraints in the `FluidLPA` cache, or creates a new `FluidLPA` via deepcopy.
`_splitSingleEdge`	`flpa_list`, `flpa_index`, `edge`	Adds `edge` as an impassable edge for the specified `FluidLPA`, and looks up the resulting set of constraints in the `FluidLPA` cache, or creates a new `FluidLPA` via deepcopy
`_queueInsert`	`flpa_list_1`, `flpa_list_2`, `flpa_cost`	Inserts the two new `FluidLPA`s into the priority queue with cost `flpa_cost`.
`_hashConstraints`	`constraints`	Sorts and hashes `constraints`.

Chapter 3

Applications to Hydraulically-Actuated 3D-Printed Robots

3.1 Introduction

Designing and building mechanisms for automated systems, including robots, is currently a labor-intensive process that requires expert-level decision making at all stages. Though computer-aided design (CAD) software is widely used during this process, these tools primarily aid during the design drafting phase. They require the user to have detailed knowledge about the materials and fabrication processes that will be used to implement the design, and critically, these material and fabrication tool choices dictate design choices that must be manually embedded within the mechanical drawings by the designer. Furthermore, once the initial design is complete, there are no guarantees that any particular part can actually be fabricated. Issues like machine tool path generation, or post-fabrication part mating clearances are typically handled as a secondary process, often requiring multiple design iterations to resolve.

As an application of MPA*, we propose an alternative design framework, tailored to the needs of roboticists and machine designers who employ kinematic linkages in their systems. Our approach allows the topology of a kinematic system to be succinctly described as set of nodes and connections, with parametrized features that allow the system to be customized via an input file based on the JSON standard. Some of these customizations are specific to the fabrication tools that these parts will be produced with - in this work an Objet Connex 260 3D printer from Stratasys Inc. (Eden Prairie, MN). However, because the designer specifies the topology, but not the geometry of the parts, the designer is freed from the burden of incorporating details of the particular fabrication approach employed into each design decision. For this reason we refer to this work as a "Mechanism Compiler".

Once the designer has created a desired linkage topology and specified it as a JSON file, our algorithms automatically create the mechanical design files in the following steps. See Figure 3-1 for a visual overview.

1. Links, actuators and the connections between them are converted to solid models via constructive solid geometry (CSG) based on parametrized primitives

2. Individual links are ordered into layers to avoid intersections

3. Hydraulic connections ("pipes") between linked hydraulic actuators are planned via MPA* and implemented via CSG

4. The surface representations of the resulting files (for both the solid and liquid materials) are exported in STL format

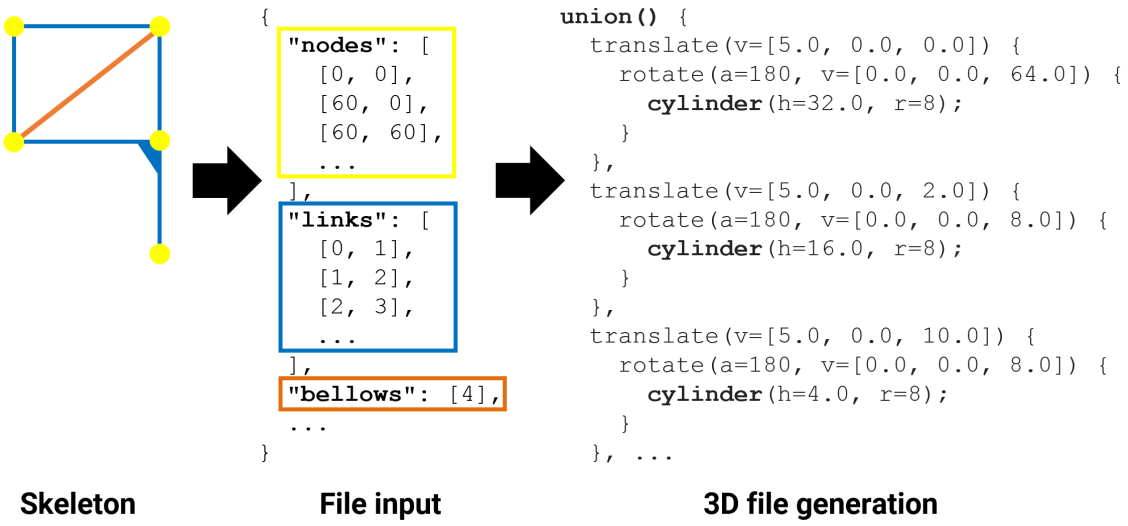


Figure 3-1: Schematic overview of robot design-automation system.

This system builds on previous work in which MacCurdy et al. showed that a commercially available 3D printer could be used to fabricate interconnected assemblies of hydraulically-actuated parts [11], however in that work the designs shown were created manually via interactive CAD tools. In related work, Coros and Thomaszewski et al. [16, 17] used an interactive design automation approach to create linkage characters with desired motions. Employing libraries of predefined parameterized modules [18, 19, 20] avoids the need for user-defined geometries, allows optimization approaches to explore the space defined by the module parameterizations, and permits complex designs to be synthesized automatically via module composition. Fuge et al. [21] demonstrated a system based on parametrized primitives that is similar to what we show here, however their framework did not include an actuator. It is worth noting that in this work we do not automate the topological design of the system of kinematic linkages, though previous work has described the synthesis of kinematic systems via the solution of analytic expressions [22], genetic algorithms [23], nonlinear programming [24], and state space methods [25].

"nodes":	List of doubles [x, y] specify the location (mm) of each node
"links":	List of doubles [n_1, n_2] specify the index into "nodes" of the two nodes that a link connects
"bellows":	List of singles l_i specify the index into "links" that will be actuated
"locks":	List of tuples [$l_1, l_2, \ldots, l_n$] specify the index into "links" of n links that will have a rigid-body constraint
"body_attachments":	List of singles n_i specify the index into "nodes" of nodes that are fixed-attachment points
"fluid_channels":	List of tuples [$n_1, n_2, \ldots, n_i$]($i = 2$ for start/end connections) that specify the index into "nodes" of any nodes that will be connected by a pipe

Table 3.1: JSON input specifications for the mechanism compiler.

3.2 Kinematic Linkage Abstraction

Our system allows the topology of an actuated kinematic system to be specified with a compact, text-based representation, and the individual geometries that implement the system are created automatically. Rather than specify the geometries that implement a desired kinematic system, the designer specifies an idealized skeleton model as a system of nodes, links that connect those nodes, and constraints on the nodes and links. This approach frees the designer from the requirement to account for design rules (e.g.: part-part clearance requirements, wall thickness, orientation constraints) in each individual part, accelerating the design process. MPA* is utilized to connect specified hydraulic fluid inputs and outputs. Future improvements on the design-automation tool could specify exclusion zones for motors, sensors, batteries, or other electronic components, and MPA* would automatically support routing hydraulic fluid pipes around these exclusion zones.

The design is formatted as a JSON-type text file, as specified in Table 3.1. Nodes are points defined in 2D space, to which links connect. Constraints on nodes and links determine the actual geometry used to implement the connections between links, what we call a "pin". Various pin types (see Figure 3-4 allow nodes to be free to move in 2D space, be fixed in 2D space (a "body attachment") or serve as fluid routing paths (a "fluid manifold"). Pin type is determined automatically, based on the constraints (including connections) on the node. Links connect nodes and are either fixed-length (Figure 3-2) or linear actuators, implemented as "bellows", following [11]. The number of bellows folds depends on the desired actuation length. Other geometric features are parametrized and exposed to the user via a configuration file. In both cases of the link or the linear actuator, the length and end-type depend on the constraints

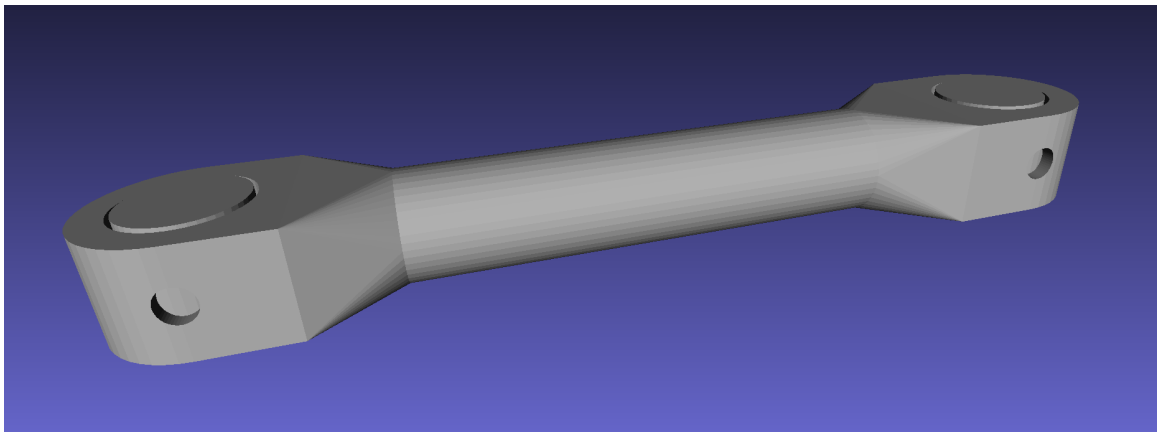


Figure 3-2: An unactuated link.

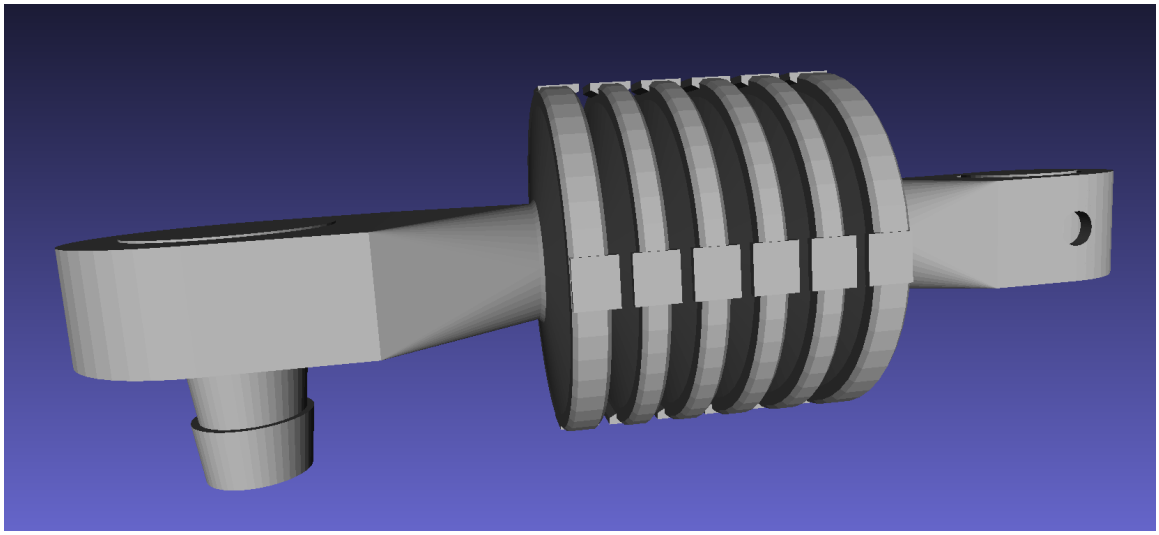


Figure 3-3: An actuated ("bellows") link.

applied to the link and the nodes that it connects to. See Figure 3-3. Links attached to the same node, by default, are unconstrained in rotation. However, two links can be "locked" together, meaning that they share a rigid body constraint. This implementation causes the links to fuse together at their ends, and impacts their layer assignment (see Section 3.3). Finally, start and end nodes for pipes connecting bellows in the system are specified with the "fluid_channels" tag, allowing automatic fluid routing that yields hydraulically-linked parts via MPA*.

The geometries that implement any particular kinematic system design are synthesized using Constructive Solid Geometry (CSG) [26, 27] methods. As we parse the JSON input file our algorithms map the specification onto a set of parametrized models for the links/bellows and pins. Specific parameters that do not depend on the topological input file (the JSON-type file) are stored in a separate geometric parameters file that contains a set of default values optimized for fabrication using the Connex 260 3D printer. Links/bellows are created on appropriate layers according the

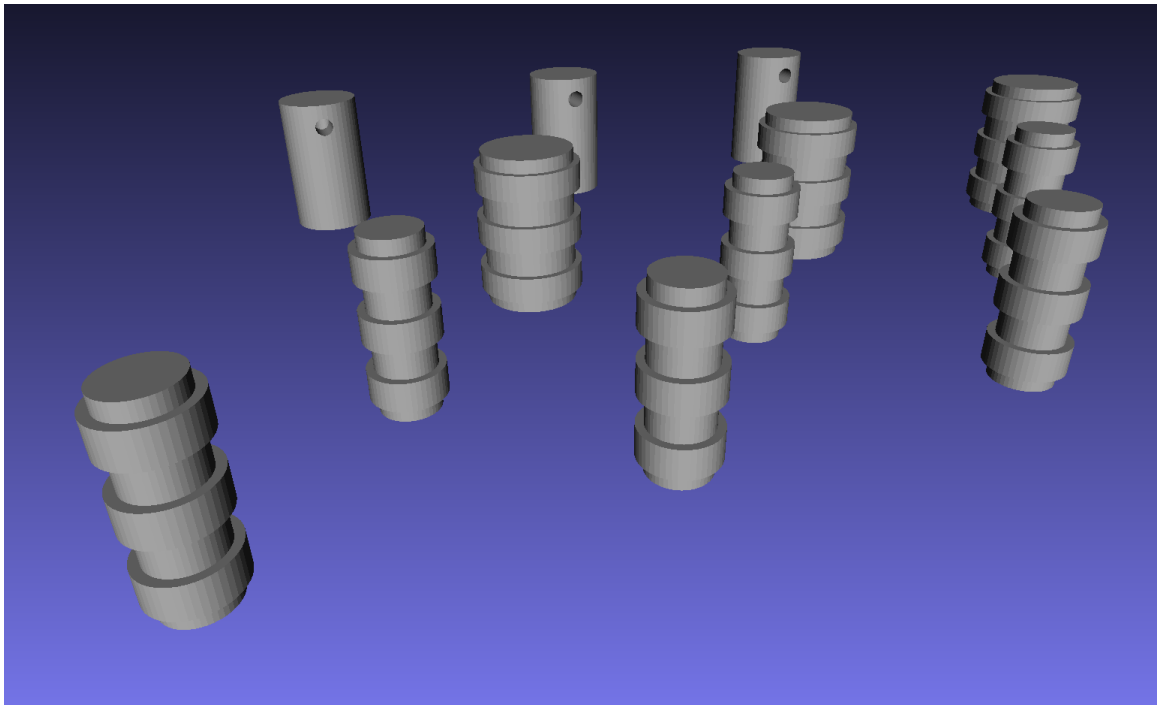


Figure 3-4: Bearing and manifold pins used in our designs.

algorithm described in Section 3.3 in order to avoid locking rotationally-unconstrained links together. The entire synthesized set of geometries are saved in an OpenSCAD (http://www.openscad.org) format. OpenSCAD and its CGAL tools (Computational Geometry Algorithms Library, http://www.cgal.org/) generate the 3D-printable surface model files (AMF or STL format).

3.2.1 CSG Pipe Implementation

Once MPA* has generated pipe-routing paths, they are converted to CSG by placing hollow spheres at the points along the path, and hollow cylinders along the edges in the path, as shown in Figure 3-5. The voids are union'd, then difference'd out so that the internal pipe is continuous.

3.3 Layer Assignment

The input specification is 2-dimensional and, by design, does not include explicit geometric information about the parts in the design. However, links attached to the same node/pin that are not meant to include a rigid-body constraint must be stacked in a layered configuration so that the physical parts do not fuse together when 3D-printed.

The space of this stacking-assignment problem is $O(l^L)$, which grows rapidly with the number of links (l) and layers (L), making exhaustive search impractical. We implemented a greedy layer-assignment algorithm, intended to yield layer stacking

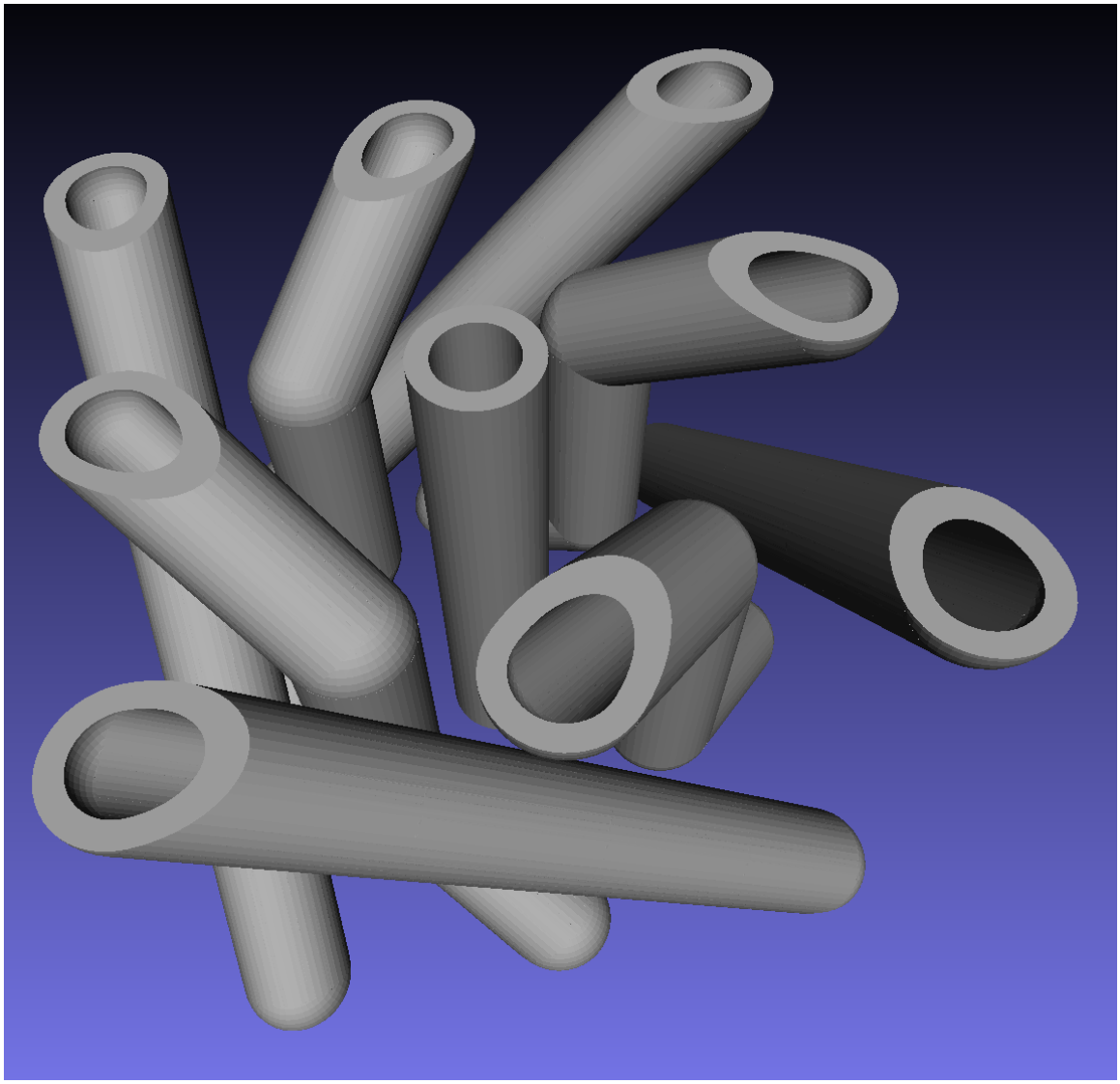


Figure 3-5: Example of 9 pipes routed via MPA*.

assignments quickly. Our layer assignment algorithm greedily assigns the lowest possible layer to links and actuators in an attempt to minimize the total height so that the final mechanism will be compact. Links which are locked together are assigned to the same layer so that their geometry will fuse. Layers are assigned to the locked links first, since these links provide more constraints. Once all links have had layers assigned, the function terminates.

Though our greedy approach has worked for all of our examples, it does not yet account for collisions between components (as might occur when a bellows link is brought close to another link), and does not provide guarantees of optimality. Coros et al [16] developed a boolean optimization-based layer assignment algorithm which could be applied to our mechanism compiler so that we can account for intersections of the bounding bounding volumes of components.

`LayerSolve()` takes as input a list of links which are not locked to other links, and

```
procedure LayerSolve(links, lockedLinks):
    for linkSet ∈ lockedLinks:
        minLayer = 0
        for link ∈ linkSet:
            minLayer = max(minLayer, MinUnusedLayer(link))
        for link ∈ linkSet:
            AssignLayer(link, minLayer)
    for link ∈ links:
        AssignLayer(link, MinUnusedLayer(link))
```

Algorithm 3: Layer assignment algorithm.

a list of link sets which are locked together. `LayerSolve()` greedily assigns layers to the locked link sets first, since they are more constrained, then fills in the remaining links at the lowest available layer.

3.4 Robot Mechanism Examples

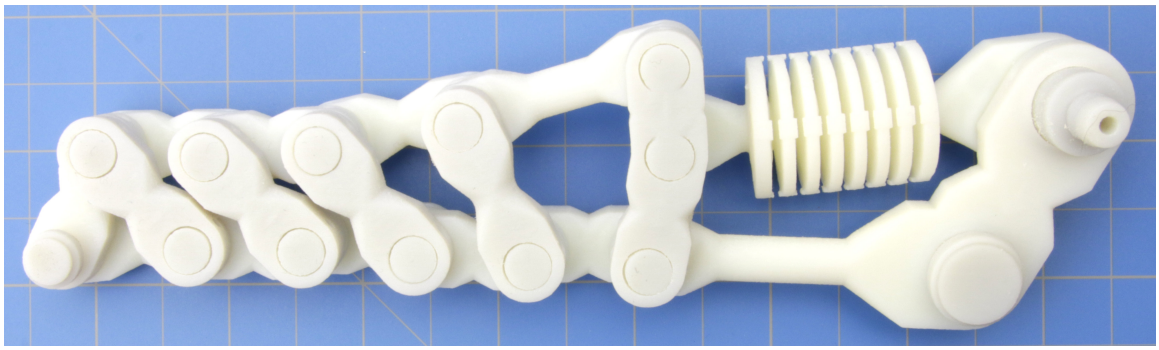
In the following, we show some examples of kinematic linkages that can be automatically generated using our mechanism compiler. Each of these designs incorporates many moving, interconnected parts, and would have required considerable effort from an experienced engineer if designed using a conventional CAD workflow. In contrast, each of these examples were conceived and printed in less than a day using our method.

3.4.1 Underactuated Finger

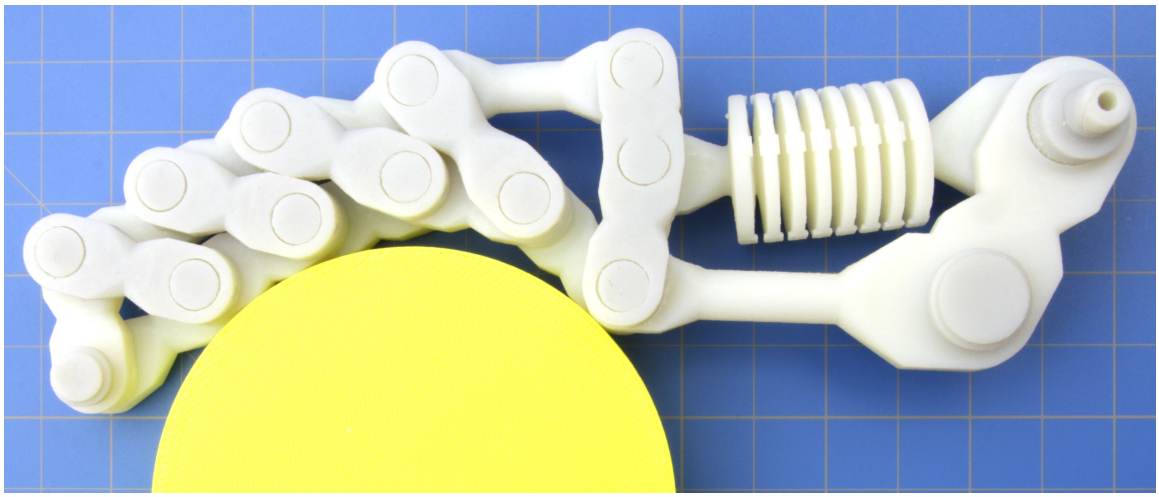
We used the kinematic compiler to design an under-actuated finger, a relatively complex kinematic linkage with 19 interconnected parts (Figure 3-6). The finger is 154 mm long in the relaxed state, and contracts to 141 mm long in the conformed state, while wrapping around an object in its workspace. This could be refined easily by modifying the linkage geometry input file. Note that a hose-attachment nozzle is visible in Figure 3-6. Our mechanism compiler automatically adds a nozzle to any body attachment points on bellows that are not connected via pipes so the bellows can be actuated via an external pressure source.

3.4.2 Leg Systems

Leg linkages are a ubiquitous feature of mobile robots. In previous work, MacCurdy et al. showed that a hydraulically-actuated hexapod robot can be automatically fabricated via 3D printing [11]. That robot employed a single DC motor that pumped fluid through the robot's body, moving the legs in sequence. However, the design process for this robot was done manually via interactive CAD tools and required many stages of iteration over the course of several weeks. By applying our new mechanism compiler, we can quickly iterate module designs for walking robots, including banks



(a) Relaxed view of an underactuated finger driven by a single printed actuator (1cm grid).



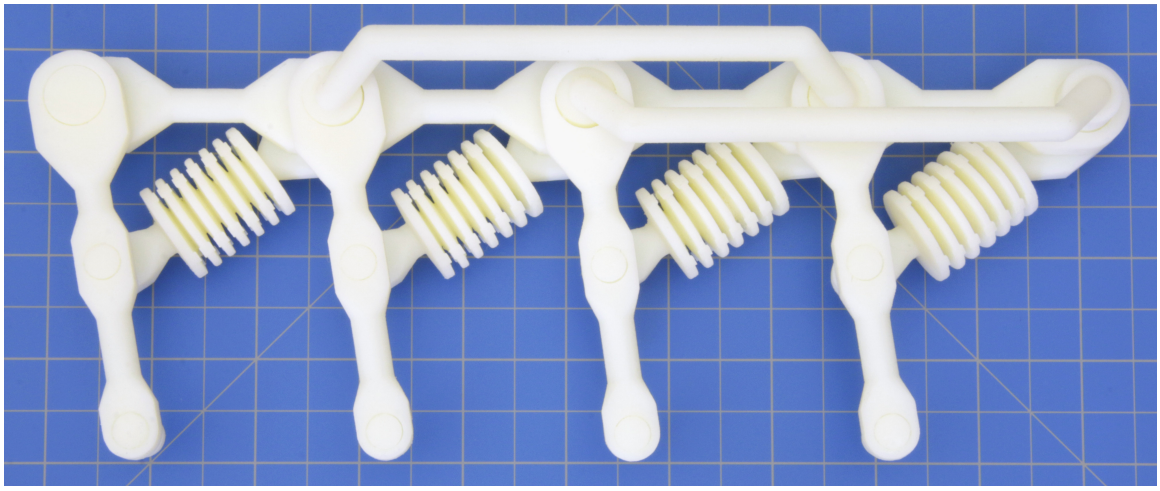
(b) Conformed view of the same finger (1cm grid).

```
{
"nodes":            [[0,0], [20,0], [-5,35], [30,25], [40,0], [55,20], [60,0], [75,18], [105,0], [-5,20], [80,0],
                    [95,18], [-72,30], [-57,0]],
"links":            [[0,1], [0,9], [2,9], [1,3], [2,3], [1,4], [3,5], [4,5], [4,6], [5,7], [6,7], [6,10], [7,11], [10,11],
                    [11,8], [10,8], [12,13], [13,0], [12,9]],
"bellows":          [18],
"locks":            [[1,2],[16,17]],
"body_attachments": [12,13],
"fluid_channels":   [ ]
}
```

(c) JSON input. Note that +X is toward the left in (a) and (b).

Figure 3-6: Underactuated finger, with corresponding JSON input.

of actuated legs. A representative example of one side of an 8-legged robot is shown in Figure 3-7 and a CAD model of both halves of the same robot, showing the hydraulic pipes connecting these sides is shown in Figure 3-8.



(a) 3D printed side of an 8-legged robot (1cm grid).


```
{
"nodes":              [[0,90], [60,90], [120,90], [180,90], [240,90], [49.6,51.3], [39.3,12.7], [109.6,51.4],
                      [99.3,12.7], [169.6,51.4], [159.3,12.7], [229.6,51.4], [219.3,12.7]],
"links":              [[0,1], [1,2], [2,3], [3,4], [1,5], [5,6], [2,7], [7,8], [3,9], [9,10], [4,11], [11,12], [0,5], [1,7],
                      [2,9], [3,11]],
"bellows":            [12,13,14,15],
"locks":              [[0,1,2,3], [4,5], [6,7], [8,9], [10,11]],
"body_attachments":   [0,1,2,3,4],
"fluid_channels":     [[0,2], [1,3]]
}
```


(b) JSON input. Note that +X is toward the left in (a).

Figure 3-7: One side of an 8-legged robot, with corresponding JSON input.

3.5 Conclusions & Limitations

Designing, fabricating and assembling the mechanical components used by robots is a time-consuming set of tasks that require expert-level planning an decision-making. Instead, we have created a system that allows a simpler text-based representation for moving kinematic assemblies. This higher-level representation is parsed by our algorithms and automatically converted into 3D design files that can be fabricated by a 3D printer. By leveraging our previous work in co-fabricating solids and liquids, these automatically created designs include fluid channels and linear actuators, allowing complete actuated mechanisms to be easily specified, automatically "compiled" into 3D representations, and automatically fabricated via multi-material 3D printing.

The system presented in this paper allows kinematic linkages to be specified in 2D, and then automatically finds a stacked 2.5D representation for the solid (3D) files so that distinct links sharing a node do not fuse together when printed. The resulting linkage assemblies are therefore approximately planar. Subsequent rotations and translations of these assemblies allows larger, more complex designs like those shown

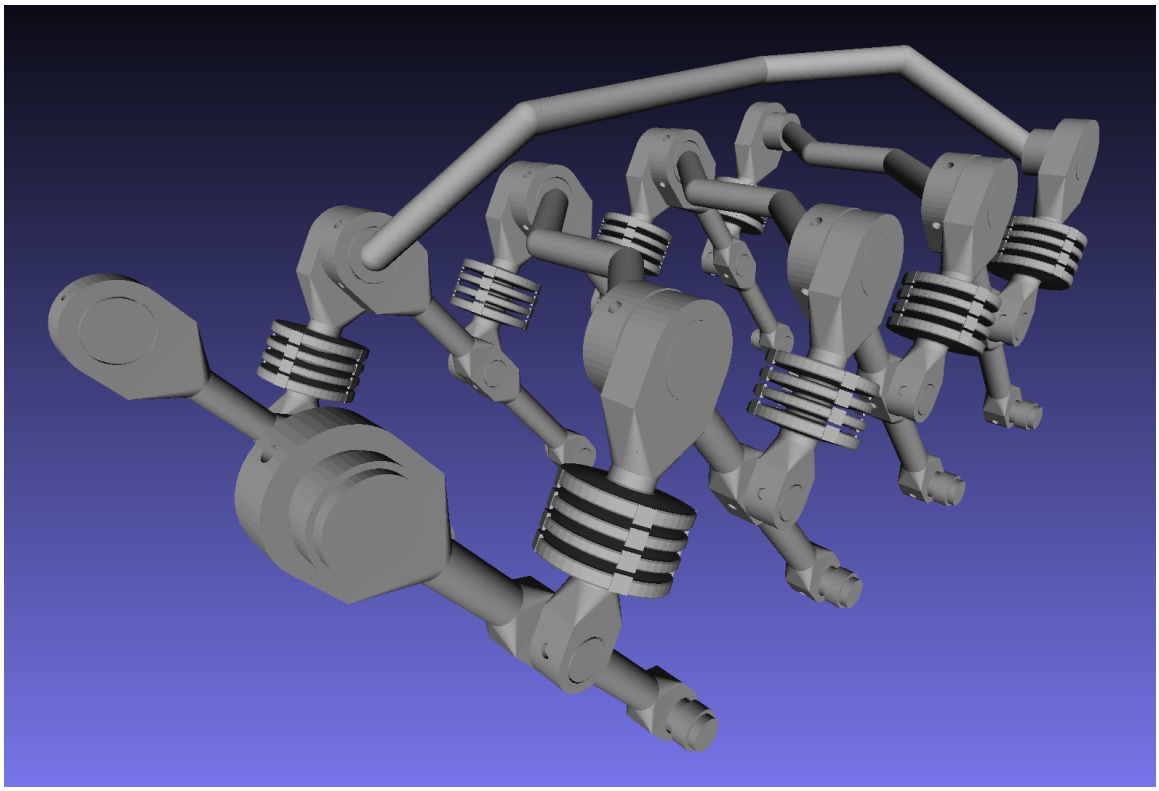


Figure 3-8: Simulated 8-legged robot with 4 pairs of actuators.

in Figure 3-8. However; in future work we plan to allow mechanisms to be specified in full 3D space, eliminating these additional steps and allowing additional design flexibility.

From a performance standpoint, OpenSCAD is by far the limiting factor in the automated design process. OpenSCAD's CGAL implementation took between 5 and 50 minutes to compile the designs shown in Figures 3-6, 3-7, and 3-8, while MPA* took less than 60 seconds to plan the fluid pipes in Figures 3-7 and 3-8. The commercial 3D printer was another limiting factor in the design iteration process, taking over a day to print and clean the examples from Figures 3-6 and 3-7.

Chapter 4

Performance of MPA*

4.1 Theoretical MPA* Performance Attributes

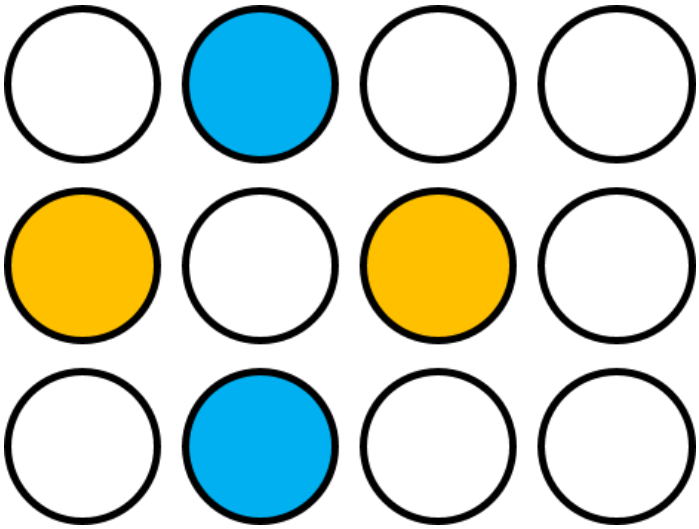
4.1.1 Sparse Grids

MPA* is particularly efficient at evaluating sparse grids. Because there is a low density of paths, there will be a low density of intersections, and the number of replans will be kept small, limiting the overall queue length. Furthermore, since each path is planned via LPA*, which uses a heuristic to perform efficient grid searches, each individual path planning or replanning operation will be at least as efficient as A*. Therefore, outside of pathological cases or dense grids, MPA*'s execution time will scale linearly with respect to the total path length, which itself scales (on average) linearly with characteristic grid dimension.

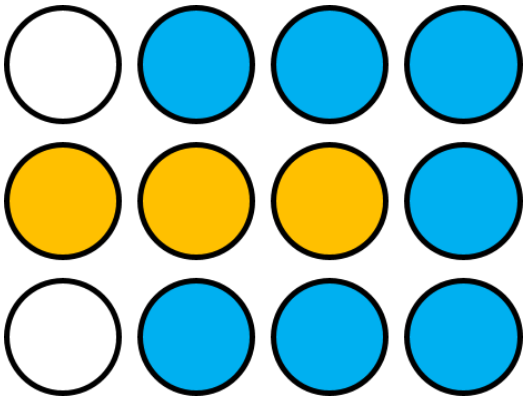
4.1.2 Compared to Greedy Algorithm

One naive solution to the circuit-routing problem is to choose a permutation of the input (start, goal) pairs, then plan (via A* or any other method) each pair in order, ensuring that each planning event takes into account previous planning events. This algorithm will be very fast, but is clearly not guaranteed to produce an optimal solution. In pathological cases or dense grids, this algorithm may not succeed at all. Consider the extremely scenario in Figure 4-1. If the blue pair is planned first, the orange pair is entirely cut off.

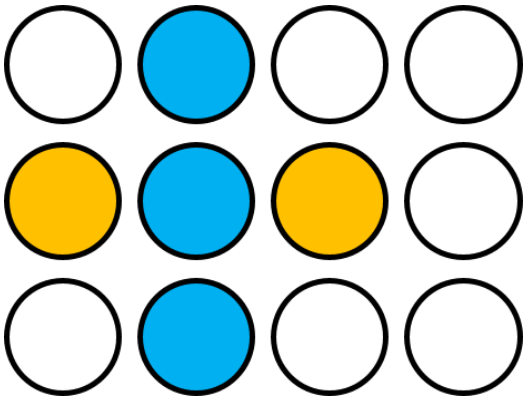
A more sophisticated variant of the greedy algorithm may try *every* permutation of input pairs. This would solve the toy example in Figure 4-1, but is still not guaranteed to produce an optimal solution or to succeed at all, especially in pathological cases or dense grids. See Figure 4-2 for a toy example where even the modified greedy algorithm does not produce an optimal solution. A slight modification to the allowed grid to prevent the vertical pipe on the edge of the volume would cause the modified greedy algorithm to fail altogether.



(a) Starting condition of a simple 2D grid.

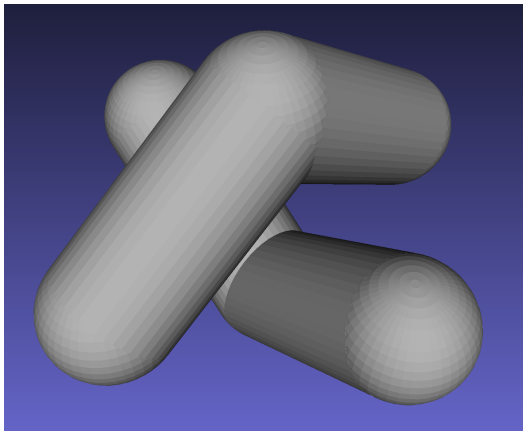


(b) Correct solution to the toy example.

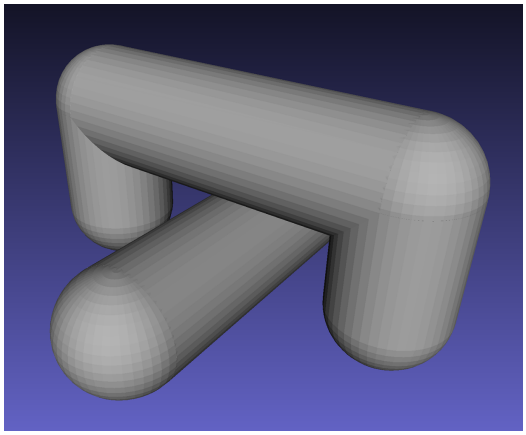


(c) Failed solution to the toy example.

Figure 4-1: Toy example of a 2D planning problem.



(a) Correct solution to the toy example.



(b) Failed solution to the toy example.

Figure 4-2: Toy example of a 3D planning problem.

4.1.3 Dense Grids

On denser grids, the benefits of the LPA* cache should be most evident, since a higher quantity of splits will inherently lead to convergent LPA*s. This is also a scenario where MPA*'s correctness guarantees are most likely to outstrip a greedy planning method, which would not be guaranteed to find an optimal solution, or indeed, any solution.

4.1.4 Flat Grids

As discussed in Section 1.1, MPA* was optimized for correctness in low-aspect-ratio grids. Heuristic and topological algorithms are better-suited for flat grid performance, and have been extremely well-optimized for circuit-routing performance.

4.2 Performance Benchmarking

In order to investigate the performance of MPA* on grids of different path densities, MPA* was run 200 times each with random non-overlapping initial (start, goal) pairs. The set of initial conditions is summarized in Table 4.1. Runs with an execution time of greater than 300 s were terminated, which represented $< 5\%$ of all runs. These tests were performed prior to the priority queue tuning and `FluidLPA` Cache additions of Sections 4.4.1 and 4.4.2, which improved speed by a total of 85 % on a representative planning task. The results of these trials are shown in Figure 4-3. As the number of paths processed increases within a certain grid size, so does the execution time, showcasing the significant impact that path density has on MPA* performance. From another point of view, larger sparse grids are solved faster than smaller dense grids, which the data also supports.

Table 4.1: Initial conditions for performance benchmarking.

Grid Characteristic Dimension	Grid Points	Number of Paths
2	27	2
4	125	2
8	729	2
16	4913	2
32	35937	2
4	125	4
8	729	4
16	4913	4
8	729	8
16	4913	8

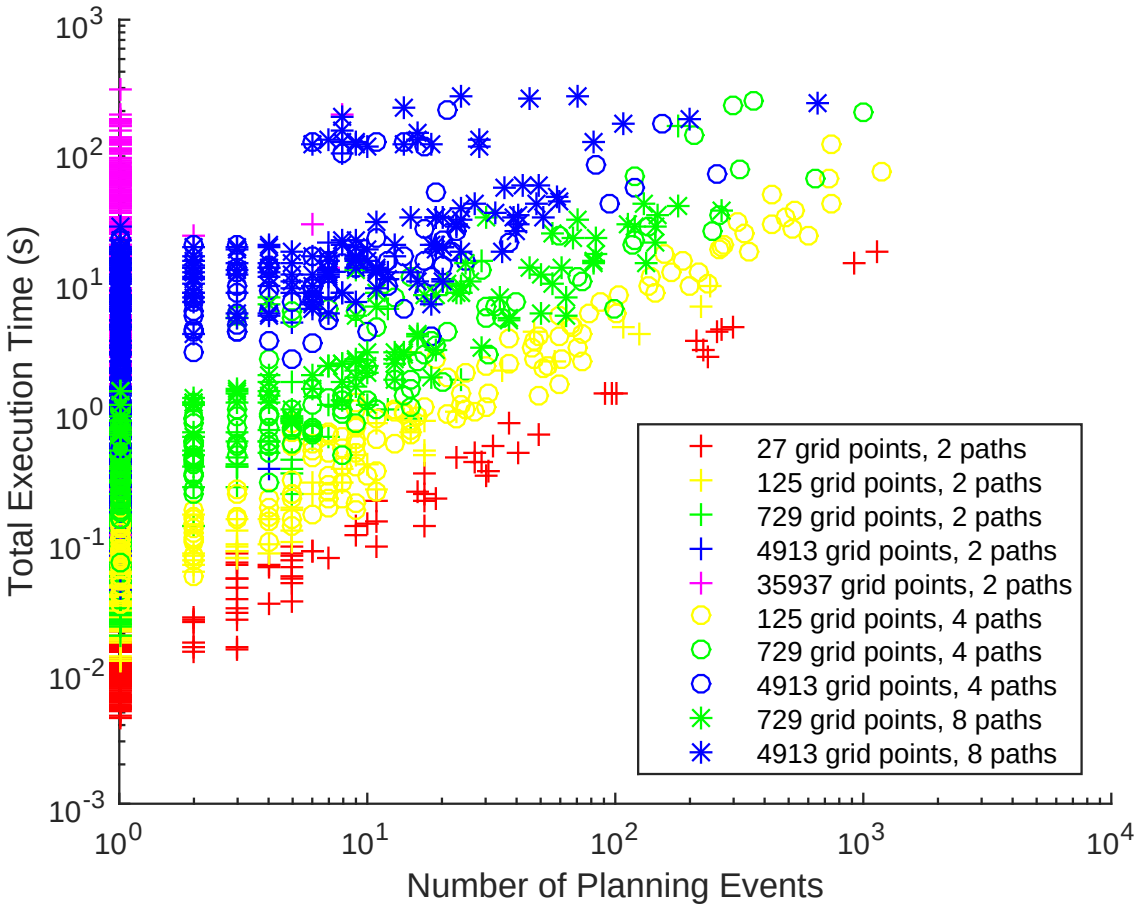


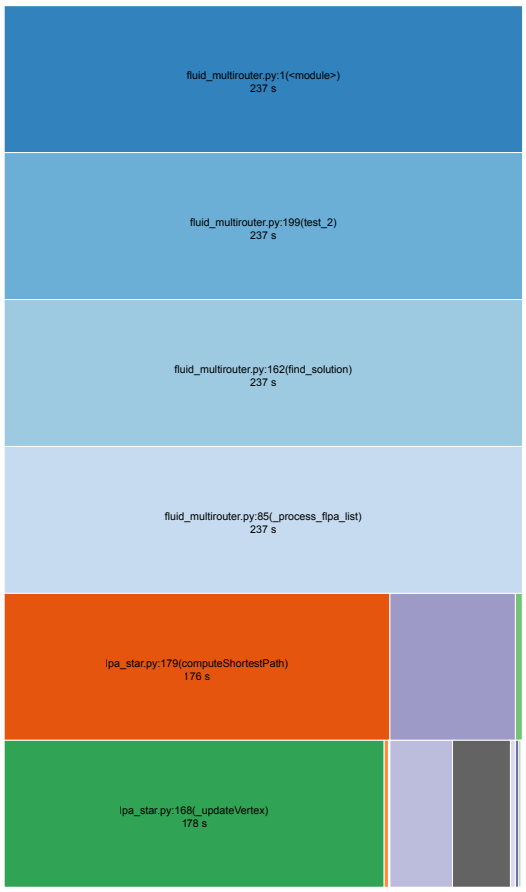
Figure 4-3: Effect of number of replans on execution time.

4.3 Performance Profiling

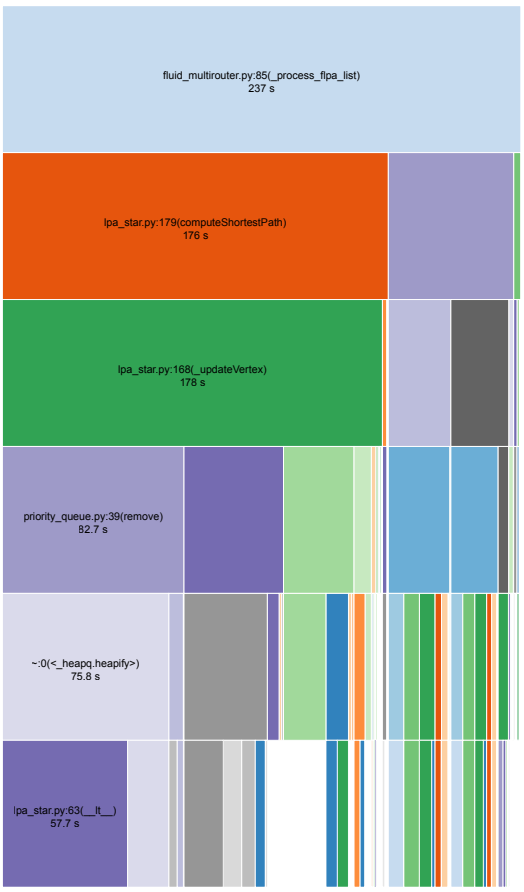
In order to focus on performance improvements and evaluate their effectiveness, the `Python` implementation of MPA* was profiled using `cProfile`, a built-in `Python` profiler [28]. Results were visualized using `snakeviz` [29]. A standard representative task was used for all profiling sessions, and a seed was given to `random` so that random data structures (just treap) would behave consistently between trials. The standard representative task is `test2()` at the bottom of `fluid_multirouter.py` (see Appendix A.1). There are 6 paths to plan, with path 1 being longest, and intersecting paths 2, 3, 4, 5, and 6 in the initial, collision-blind solution. The exact paths and correct solutions are in Table 4.2. An example profiling result is shown in Figure 4-4.

Table 4.2: Paths used for performance profiling task

Start	Goal	Correct Path
(1, 0, 1)	(1, 6, 1)	(1, 0, 1), (1, 0, 2), (1, 1, 2), (1, 2, 2), (1, 3, 2), (1, 4, 2), (1, 5, 2), (1, 6, 2), (1, 6, 1)
(0, 1, 1)	(2, 1, 1)	(0, 1, 1), (1, 1, 1), (2, 1, 1)
(0, 2, 1)	(2, 2, 1)	(0, 2, 1), (1, 2, 1), (2, 2, 1)
(0, 3, 1)	(2, 3, 1)	(0, 3, 1), (1, 3, 1), (2, 3, 1)
(0, 4, 1)	(2, 4, 1)	(0, 4, 1), (1, 4, 1), (2, 4, 1)
(0, 5, 1)	(2, 5, 1)	(0, 5, 1), (1, 5, 1), (2, 5, 1)



(a) Profiling result for a Binomial Heap Priority Queue.



(b) Relevant section of profiling result for a Binomial Heap Priority Queue.

Figure 4-4: Profiling results for MPA* with Binomial Heap Priority Queue.

4.4 Performance Tuning

4.4.1 Priority Queue Tuning

Each of the data structures mentioned in Section 2.3.2 was implemented, and profiled using the standard representative task. The results are summarized in Table 4.3. The results were in good agreement with predictions from asymptotic behavior. Commentary for each result is provided below

- Binomial heap augmented with hashtable, using `Python`'s `heapq` and `set`. This was the first version implemented, and was used for initial benchmarking. This had $O(1)$ `insert`, $O(\log n)$ `pop`, and $O(n)$ `remove`. Performance was significantly impacted by the linear-time `remove`. See Appendix B-1 for full profiling results.

- Linked list augmented with hashtable, with elements of the hashtable pointing to elements of the linked list. This allowed for $O(1)$ `pop` and `remove`, but $O(n)$ `insert`. Performance was significantly impacted by the linear-time `insert`, but it was a factor of 2 faster than `remove` from the binomial heap. See Appendix B-2 for full profiling results.

- Array augmented with hashtable. Though this only had $O(n \log n)$ pop and $O(n)$ remove, with $O(1)$ `insert` and with excellent amortized performance from `Python`'s built-in `list` and `sort`, this still performed reasonably well. Each of `insert` and `remove` were a factor of 2 faster than `remove` from the binomial heap, so performance was equivalent overall. See Appendix B-3 for full profiling results.

- Treap augmented with hashtable, using `treap` [14]. This was the first high-performance data structure used, with $O(\log n)$ performance on all relevant methods. Due to the improved asymptotic complexity, this was the fastest method measured. The vast majority of time was spent in LPA*. See Appendix B-4 for full profiling results.

- Nested lazy-length lists augmented with hashtable, using `sortedcontainers` [15]. This is a particularly interesting one. Although asymptotically slower than a treap or Red-Black tree, the nested lazy-length lists take advantage of `Python`'s extremely well-optimized list performance in the same manner that the array augmented with a hashtable did, and performed better in practice, and had a (constant factor) smaller memory footprint. Performance was nearly as fast as the treap implementation, but `deepcopy` performance was significantly degraded. However, because it was written in pure Python, caching was easily implemented and improved performance beyond the treap implementation. See Section 4.4.2 for discussions on adding the cache. See Appendix B-5 for full profiling results.

Table 4.3: Performance Tuning Results (times in seconds)

Priority Queue Data Structure	Caching	Total	`insert`	`pop`	`remove`	`deepcopy`
Binomial Heap	-	237	0.986	0.2391	82.74	49.70
Linked List	-	205	39.5	0.3675	1.058	46.04
Array	-	237	45.44	0.03587	43.20	44.89
Treap	-	73.0	1.78	0.01483	1.635	6.95
Nested Lazy-Length Lists	-	227	3.973	0.06836	3.002	137.3
Nested Lazy-Length Lists	Yes	36.5	1.019	0.02426	1.051	4.081

4.4.2 `FluidLPA` Cache

As mentioned in Section 4.4.1, the nested lazy-length lists performed almost as well as treaps except in `deepcopy` performance. One way to both reduce the number of `deepcopy` calls and the number of LPA* calls (the other dominating factor in overall performance) is to implement an LPA* caching layer. See Section 2.1.3 for a deeper discussion on the details behind implementing a caching layer. After implementing the caching layer, `deepcopy` time improved by over 97%, and the number of LPA* calls was roughly cut by 60%. This resulted in an overall performance increase of 84%, or an improvement of 50% over the treap implementation. See Table 4.3 for summarized profiling results and Appendix B-6 for full profiling results.

The dominating factors in overall performance after the caching layer was implemented were low-level computational tasks of LPA*, such as calculating the cost and heuristic functions, and lexical key comparisons. These functions are already constant-time, so the most significant performance improvements remaining for MPA* would only be realized by re-implementing MPA* in a more performant language, such as C++, or by paralellizing MPA*.

4.5 Memory Efficiency

MPA* is also reasonably memory efficient. LPA* was implemented with lazy grid expression - grid points are not initialized until they are explored for the first time. Since LPA* uses heuristics, the number of grid points which need to be explored is small with respect to the search space. Furthermore, MPA* can search the constraint space in a lazy manner - splits are only created as they are explored. Further memory improvements were made by maintaining a cache of LPA* objects, since at each branch operation, only one LPA* object out of the list is modified for each of the two branches. In practice, our `Python` implementation of lazy-grid LPA* and MPA* consumes on the order of 10 MB of memory for searches of a path density and grid resolution appropriate for our fluid-routing needs discussed in Section 3. Even in the larger grids with more paths that we evaluated for benchmarking purposes, memory usage stayed under 1 GB. Overall, although memory efficiency was considered in the design and implementation of MPA*, it was not heavily benchmarked or optimized in this iteration.

Chapter 5

Conclusions

5.1 Results

This thesis identified a gap in current work on the circuit-routing problem, namely for low aspect ratio cases where an exact solution is desired. A new algorithm, MPA*, to solve the circuit-routing problem is proposed, shown to be correct, and implemented. As a subproblem, LPA* was implemented for the first time in `Python` and for the first time for non-2D grids. MPA* is optimized for low aspect ratio instances of the circuit-routing problem. Applications in hydraulically-actuated 3D-printed robots are explored, and robots designed in part via MPA* are physically printed. Finally, performance is analyzed and tuned such that any large future performance gains would likely be restricted to reimplementation in a more performant language than `Python`, or parallelization. Total performance gains are about 85% better the slowest reasonable implementation.

5.2 Discussion & Future Work

MPA* is fairly unique among algorithms which solve the circuit-routing problem because it provides the exactly-correct solution, and this is the primary motivating feature. In hydraulically-actuated 3D printed robot design, the number of times MPA* will be called is very low, even if the designer decides to go through a number of iterations. In practice, MPA* is not the bottleneck in 3D printed robot design - compiling the CSG is more time-consuming by about an order of magnitude (anywhere between 5 and 50 minutes), and 3D-printing the robots is more time-consuming still by more than another order of magnitude (anywhere between 5 and 30 hours). Therefore, MPA* is an excellent candidate for use in 3D printed robot design. Further explorations of MPA*'s applications in other fields are certainly merited. MPA* could also be used as a benchmarking tool for heuristic-based circuit-routing algorithms, to determine how far away from an optimal solution they are.

A few remaining opportunities for performance improvements are proposed:

- Exploration of nodes in the outer graph search and LPA* solutions in the inner searches can both be parallelized. Paralellizing LPA* solutions within the same

`LPAlist` is trivially easy, but due to the caching layer implemented, would not likely provide much speedup except in the initial round.

- Once candidate solutions have been found, solutions or intermediate states which are obviously non-optimal can be discarded from the priority queue. They would never have been explored, but this change will improve memory efficiency.
- In conjunction with the above, a naive solution can be used to seed the candidate solutions, again, to improve memory efficiency.
- Collision-checking between paths could be implemented via a bloom filter, which would improve runtime of the position-collision-detection subroutine by a factor of n, the total number of points in all paths. However, this subroutine already takes less than 10 ms total, out of the 36.5 s total runtime, so any improvements are largely irrelevant for overall performance.

Overall, MPA* is functional and solves a niche problem which other algorithms did not adequately address.

Appendix A

Code

A.1 fluid_multirouter.py

```
"""Routes multiple fluid channels simultaneously."""

import copy
import numpy as np
import lpa_fluid_router as flpa
import random
import lpa_math

import priority_queue as pq

random.seed(1)

class MPA(object):
    """BFS Python implementation."""
    def __init__(self, desired_routes, valid_region_func, debug=False):
        """Args:
            desired_routes: List of (startPos, endPos)
            valid_region_func: Function taking a position as input and returning
                True or False depending on whether this position is in an
                allowed region.
        """

        # Initialize variables
        self.desired_routes = desired_routes
        self.debug = debug

        # Initialize starting FLPA
        flpa_list = []
        for route in desired_routes:
            state_lookup_dict = {}
            fluid_state_factory = flpa.StateFactory(
                flpa.NodeState, state_lookup_dict, valid_region_func)
            s_start = fluid_state_factory.makeOrGetStateByPos(route[0])
            s_goal = fluid_state_factory.makeOrGetStateByPos(route[1])

            route_flpa = flpa.FluidLPA(s_start, s_goal, fluid_state_factory,
    state_lookup_dict)

            flpa_list.append(route_flpa)

        self.flpa_queue = pq.Queue()
        self.flpa_queue.insert((-1,tuple(flpa_list)))
        self._flpa_cache = {}

    def _checkEdgeCollision(self, edge_1, edge_2):
```

```
        a0 = np.array(edge_1[0].pos)
        a1 = np.array(edge_1[1].pos)
        b0 = np.array(edge_2[0].pos)
        b1 = np.array(edge_2[1].pos)
        if lpa_math.closestDistanceBetweenLines(a0, a1, b0, b1) < 0.8:
            if self.debug:
                print ('Edge collision detected, distance is %s' %
                    lpa_math.closestDistanceBetweenLines(a0, a1, b0, b1))
            return True
        return False


    def _checkCollisionFree(self, flpa_list, flpa_index_1, flpa_index_2,
                            path_1, path_2, flpa_cost):
        """Checks that path_1 and path_2 are collision-free. If they are not,
        calls a split method appropriately and returns False."""

        # check direct collisions
        for node in path_1:
            if node in path_2:
                if self.debug:
                    print "node overlap conflict at %s" % (node.pos,)
                self._splitFlpaPos(flpa_list, flpa_index_1, flpa_index_2,
                                   node, flpa_cost)
                return False

        # check edge collisions
        edges_1 = [(path_1[i], path_1[i+1]) for i in range(len(path_1) - 1)]
        edges_2 = [(path_2[i], path_2[i+1]) for i in range(len(path_2) - 1)]
        for edge_1 in edges_1:
            for edge_2 in edges_2:
                if self._checkEdgeCollision(edge_1, edge_2):
                    if self.debug:
                        print ("edge conflict at (%s, %s), (%s, %s)" %
                            (edge_1[0].pos, edge_1[1].pos, edge_2[0].pos,
                             edge_2[1].pos))
                    self._splitFlpaEdge(flpa_list, flpa_index_1, flpa_index_2,
                                        edge_1, edge_2, flpa_cost)
                    return False

        # no collisions
        return True


    def _processFlpaList(self, flpa_list):
        """Computes the shortest path for all of the FLPA's in flpa_list. Splits
        on a random bad position and returns None if the flpa_list is
        unsolvable. Returns a list of paths if the flpa_list is solvable."""

        paths = []
        total_flpa_cost = 0
        for route_flpa in flpa_list:
            if self.debug:
                print "Solving new FLPA: from %s to %s" % (route_flpa.sStart,
route_flpa.sGoal)
                route_flpa.printConstraints()
            route_flpa.computeShortestPath()
            if self.debug: print "computed shortest path"
            (tmp_path, cost) = route_flpa.getShortestPath()
            if self.debug: print "got shortest path"
            if tmp_path is None:
                if self.debug: print "path is none"
                return (None, float("inf"))
            if self.debug: print "appending"
            paths.append(tmp_path)
            if self.debug: print "appended"
            total_flpa_cost += cost
            if self.debug: print "added"
```

```
    # collision check
    if self.debug:
        print "performing collision check"
        for path in paths:
            print "Potential path is:"
            for node in path:
                print node.pos
    for i in range(len(paths)):
        for j in range(i+1, len(paths)):
            if self._checkCollisionFree(flpa_list, i, j, paths[i],
                                        paths[j], total_flpa_cost):
                continue
            else:
                if self.debug: print "returning nothing"
                return (None, float("inf"))
    if self.debug: print "actually returning something"
    return (paths, total_flpa_cost)


def _splitFlpaEdge(self, flpa_list, flpa_index_1, flpa_index_2, edge_1,
                   edge_2, cost):
    """Splits two routes at edge_1 and edge_2 respectively and appends the
    new search objects to the flpa_queue."""
    left_flpa_list = list(flpa_list)
    right_flpa_list = list(flpa_list)

    left_flpa_list = self._splitSingleEdge(left_flpa_list, flpa_index_1, edge_1)
    right_flpa_list = self._splitSingleEdge(right_flpa_list, flpa_index_2, edge_2
)
    self._queueInsert(left_flpa_list, right_flpa_list, cost)


def _splitSingleEdge(self, flpa_list, flpa_index, edge):
    flpa_constraints = flpa_list[flpa_index].getConstraints() # (node, edge)
    flpa_constraints[1].add(edge)
    h = self._hashConstraints(flpa_constraints)
    flpa_constraints[1].remove(edge)
    if h in self._flpa_cache:
        if self.debug:
            print("cache hit")
        new_flpa = self._flpa_cache[h]
    else:
        if self.debug:
            print("cache miss")
        new_flpa = copy.deepcopy(flpa_list[flpa_index])
        new_flpa.makeEdgeImpassable(edge)
        self._flpa_cache[h] = new_flpa
    if self.debug:
        print("hashtable has %s elements" % (len(self._flpa_cache),))
    flpa_list[flpa_index] = new_flpa

    return tuple(flpa_list)


def _splitFlpaPos(self, flpa_list, flpa_index_1, flpa_index_2, bad_node,
                  cost):
    """Splits two routes at bad_pos and appends the new search objects to
    the flpa_queue."""

    left_flpa_list = list(flpa_list)
    right_flpa_list = list(flpa_list)

    left_flpa_list = self._splitSinglePos(left_flpa_list, flpa_index_1, bad_node)
    right_flpa_list = self._splitSinglePos(right_flpa_list, flpa_index_2,
bad_node)
    self._queueInsert(left_flpa_list, right_flpa_list, cost)
```

```
    def _splitSinglePos(self, flpa_list, flpa_index, bad_node):
        flpa_constraints = flpa_list[flpa_index].getConstraints() # (node, edge)
        flpa_constraints[0].add(bad_node)
        h = self._hashConstraints(flpa_constraints)
        flpa_constraints[0].remove(bad_node)
        if h in self._flpa_cache:
            if self.debug:
                print("cache hit")
            new_flpa = copy.deepcopy(self._flpa_cache[h])
        else:
            if self.debug:
                print("cache miss")
            new_flpa = copy.deepcopy(flpa_list[flpa_index])
            new_flpa.makeNodeImpassable(new_flpa.state_factory.makeOrGetStateByPos(
bad_node.pos))
            self._flpa_cache[h] = new_flpa
        if self.debug:
            print("hashtable has %s elements" % (len(self._flpa_cache),))
        flpa_list[flpa_index] = new_flpa

        return tuple(flpa_list)

    @staticmethod
    def _hashConstraints(constraints):
        nodes = (node.pos for node in sorted(constraints[0]))
        edges = ((edge[0].pos, edge[1].pos) for edge in sorted(constraints[1]))
        toReturn = hash((tuple(nodes), tuple(edges)))
        return toReturn

    def _queueInsert(self, flpa_list_1, flpa_list_2, flpa_cost):
        """Inserts into the priority queue with appropriate cost.
        """
        self.flpa_queue.insert((flpa_cost, tuple(flpa_list_1)))
        self.flpa_queue.insert((flpa_cost, tuple(flpa_list_2)))


    def findSolution(self):
        minCost = float("inf")
        best = None
        while minCost > self.flpa_queue.topKey()[0]:
            tmp_flpa_list = self.flpa_queue.pop()[1]
            if self.debug:
                print("Starting new FLPA. The current queue length is %s" %
                      len(self.flpa_queue))
            (paths, totalcost) = self._processFlpaList(tmp_flpa_list)
            minCost = min(minCost, totalcost)
            if minCost == totalcost:
                best = paths
                if self.debug: print("\t valid candidate solution")
        return (best, minCost)



def test_1():
    """Tests multi-object fluid routing."""
    route_1 = ((0, 0, 1), (2, 2, 1))
    route_2 = ((2, 0, 1), (0, 2, 1))
    routes = [route_1, route_2]

    def fluid_is_valid(pos):
        return max(pos) <= 2 and min(pos) >= 0

    bfs_obj = MPA(routes, fluid_is_valid, debug=True)

    (paths, cost) = bfs_obj.findSolution()
    print "\n\n\n"
    for path in paths:
```

```
        for state in path:
            print state.pos
        print ""

def test_2():
    """Another test for multi-object fluid routing."""
    route_main = ((1, 0, 1), (1, 6, 1))
    route_cross_1 = ((0, 1, 1), (2, 1, 1))
    route_cross_2 = ((0, 2, 1), (2, 2, 1))
    route_cross_3 = ((0, 3, 1), (2, 3, 1))
    route_cross_4 = ((0, 4, 1), (2, 4, 1))
    route_cross_5 = ((0, 5, 1), (2, 5, 1))

    routes = [route_main, route_cross_5, route_cross_4, route_cross_3,
              route_cross_2, route_cross_1]

    # routes = [route_main, route_cross_5]

    def fluid_is_valid(pos):
        return max(pos) <= 10 and min(pos) >= 0

    bfs_obj = MPA(routes, fluid_is_valid, debug=False)
    (paths, cost) = bfs_obj.findSolution()
    print "\n\n\n"
    for path in paths:
        for state in path:
            print state.pos
        print ""


if __name__ == '__main__':
    # test_1()
    test_2()
```

A.2 lpa_fluid_router.py

```
"""Simultaneous fluid channel routing in 3D-space."""

import lpa_star
import copy

#pylint: disable=attribute-defined-outside-init


class NodeState(lpa_star.State):
    """State subclass which defines nodes for fluid routing."""
    def setNodeLookupDict(self, node_lookup_dict):
        self.lookup_dict = node_lookup_dict

    def setValidPosLookupFunc(self, pos_f):
        self.is_valid_pos_f = pos_f

    def pred(self):
        preds = []
        for i in range(-1, 2):
            for j in range(-1, 2):
                for k in range(-1, 2):
                    if (i, j, k) == (0, 0, 0):
                        continue
                    possible_pos = (self.pos[0] + i, self.pos[1] + j, self.pos[2] + k
    )
                    if self.is_valid_pos_f(possible_pos):
                        preds.append(possible_pos)
        return preds

    def succ(self):
```

```
        return self.pred()

    def __copy__(self):
        result = NodeState(self.pos[:], self.k[:])
        result.setValidPosLookupFunc(self.is_valid_pos_f)
        return result

    def __deepcopy__(self, memo):
        # raise Exception
        result = NodeState(self.pos[:], self.k[:])
        memo[id(self)] = result
        result.setNodeLookupDict(copy.deepcopy(self.lookup_dict, memo))
        result.setValidPosLookupFunc(self.is_valid_pos_f)

        return result


class FluidLPA(lpa_star.LPA):
    """LPA* subclass which defines the heuristic and cost functions for fluid
    LPA*.
    """
    def _h(self, s, s_goal):
        return self._c(s, s_goal)

    def _c(self, s, s2):
        if s in self._impassable_nodes or s2 in self._impassable_nodes:
            return float("inf")
        if (s, s2) in self._impassable_edges or (s2, s) in self._impassable_edges:
            return float("inf")
        p_1 = s.pos
        p_2 = s2.pos
        return sum([(p_1[i] - p_2[i])**2 for i in range(3)])**0.5

    def makeNodeImpassable(self, impassable_node):
        self._impassable_nodes.add(impassable_node)
        for pos in impassable_node.pred():
            node = self.state_factory.makeOrGetStateByPos(pos)
            self._updateVertex(node)

    def makeEdgeImpassable(self, impassable_edge):
        self._impassable_edges.add(impassable_edge)
        for node in impassable_edge:
            self._updateVertex(node)


class StateFactory(object):
    """Factory for creating or getting states."""
    def __init__(self, state_class, node_lookup_dict, valid_pos_lookup_func,
                 debug=False):
        self.state_class = state_class
        self.node_lookup_dict = node_lookup_dict
        self.valid_pos_lookup_func = valid_pos_lookup_func
        self.debug = debug

    def makeOrGetStateByPos(self, pos):
        """Looks up a position in the dictionary, returns the state if the pos
        exists in the dictionary. Creates the state and adds it to the dict
        if it does not already exist, then returns it."""
        if self.debug:
            print "got request for pos %s" % (pos,)
        tmp_state = self.state_class(pos, (float("inf"), float("inf")))
        if tmp_state in self.node_lookup_dict:
            if self.debug:
                print "found state in dict: %s" % (self.node_lookup_dict[tmp_state],)
            return self.node_lookup_dict[tmp_state]
        elif self.valid_pos_lookup_func(pos):
            if self.debug:
                print "didn't find state, making new one"
```

```
            new_s = self.state_class(pos, (float("inf"), float("inf")))
            new_s.setNodeLookupDict(self.node_lookup_dict)
            new_s.setValidPosLookupFunc(self.valid_pos_lookup_func)
            self.node_lookup_dict[new_s] = new_s
            return new_s
        else:
            # The requested state is not a valid position.
            return None

    def updateState(self, new_state):
        self.node_lookup_dict[new_state] = new_state


def fluid_is_valid(pos):
    """Determines whether a given position is valid or not."""
    return max(pos) <= 10 and min(pos) >= 0

def fluid_is_valid_2(pos):
    """Determines whether a given position is valid or not."""
    return min(pos) >= 0 and pos[0] <= 2 and pos[1] < 1 and pos[2] < 1


def test1():
    """Tests the fluid routing. Also a helpful example of usage."""
    state_lookup_dict = {}
    fluid_state_factory = StateFactory(NodeState, state_lookup_dict,
                                       fluid_is_valid, debug=False)

    s_start = fluid_state_factory.makeOrGetStateByPos((0, 0, 0))
    s_goal = fluid_state_factory.makeOrGetStateByPos((2, 2, 2))

    flpa = FluidLPA(s_start, s_goal, fluid_state_factory, state_lookup_dict)

    flpa.computeShortestPath()
    (path, cost) = flpa.getShortestPath()

    flpa.computeShortestPath()
    (path1, cost) = flpa.getShortestPath()

    flpa2 = copy.deepcopy(flpa)
    flpa2.makeNodeImpassable(
        flpa2.state_factory.makeOrGetStateByPos((1, 1, 1)))

    flpa2.computeShortestPath()
    assert path1 != flpa2.getShortestPath()[0]

    flpa.computeShortestPath()
    assert path1 == flpa.getShortestPath()[0]

    for state in flpa.getShortestPath()[0]:
        print state.pos

    print ""

    for state in flpa2.getShortestPath()[0]:
        print state.pos

def test2():
    """Tests that fluid routing can deal with impassable routes."""
    state_lookup_dict = {}
    fluid_state_factory = StateFactory(NodeState, state_lookup_dict,
                                       fluid_is_valid_2, debug=False)

    s_start = fluid_state_factory.makeOrGetStateByPos((0, 0, 0))
    s_goal = fluid_state_factory.makeOrGetStateByPos((2, 0, 0))

    flpa = FluidLPA(s_start, s_goal, state_lookup_dict, debug=True)
    flpa.makeNodeImpassable(
```

```
        flpa.state_factory.makeOrGetStateByPos((1, 0, 0)))

    flpa.computeShortestPath()
    print "computed shortest path"
    (path, cost) = flpa.getShortestPath()
    assert path is None
    assert cost == float("inf")

def test3():
    """Tests the fluid routing. Also a helpful example of usage."""
    lpa_lookup_dict = {}
    state_lookup_dict = {}
    fluid_state_factory = StateFactory(NodeState, state_lookup_dict,
                                       fluid_is_valid, debug=False)

    s_start = fluid_state_factory.makeOrGetStateByPos((0, 0, 0))
    s_goal = fluid_state_factory.makeOrGetStateByPos((2, 2, 2))

    flpa = FluidLPA(s_start, s_goal, fluid_state_factory, state_lookup_dict)

    flpa.computeShortestPath()
    (path1, cost) = flpa.getShortestPath()
    lpa_lookup_dict[_hashConstraints(flpa.getConstraints())] = flpa

    flpa2 = copy.deepcopy(flpa)
    flpa2.makeNodeImpassable(
        flpa2.state_factory.makeOrGetStateByPos((1, 1, 1)))

    flpa2.computeShortestPath()
    assert path1 != flpa2.getShortestPath()[0]
    lpa_lookup_dict[_hashConstraints(flpa2.getConstraints())] = flpa

    flpa3 = copy.deepcopy(flpa)
    flpa3.makeNodeImpassable(
        flpa3.state_factory.makeOrGetStateByPos((1, 1, 1)))

    assert flpa3 in lpa_lookup_dict
    flpa4 = lpa_lookup_dict[flpa3]
    assert flpa2 == flpa4

    flpa.computeShortestPath()
    assert path1 == flpa.getShortestPath()[0]

    for state in flpa.getShortestPath()[0]:
        print state.pos

    print ""

    for state in flpa2.getShortestPath()[0]:
        print state.pos

def _hashConstraints(constraints):
    nodes = (node.pos for node in sorted(constraints[0]))
    edges = ((edge[0].pos, edge[1].pos) for edge in sorted(constraints[1]))
    return hash((nodes, edges))

if __name__ == '__main__':
    test1()
    test2()
    test3()
```

A.3 lpa_star.py

```
"""LPA* implementation in Python."""
```

```
import priority_queue_sortedset as pq

# c: cost
# g*(s): dist from start to s
# h(s, s_goal): heuristic (nonnegative, obey triangle ineq)
# g(s): estimate of g*(s)
# rhs(s): 0 if s is start, min route to s from pred(s) otherwise

# k1 = min(g(s), rhs(s) + h(s, sgoal)) (travel cost to start + heuristic to goal)
# k2 = min(g(s), rhs(s))               (travel cost to start approx)

# overconsistent: g(s) > rhs(s)  --> set g(s) = rhs(s)
# underconsistent: g(s) < rhs(s) --> set g(s) = inf

class PathTooLongException(Exception):
    pass

class State():
    """An arbitrary-dimension state for LPA*"""

    def __init__(self, pos, k):
        """Sets position and k variables"""
        self.pos = pos
        self.k = k
        assert isinstance(self.k, tuple)
        assert len(self.k) == 2
        assert isinstance(self.pos, tuple)

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return other.pos == self.pos
        return NotImplemented

    def __ne__(self, other):
        res = self.__eq__(other)
        if res is NotImplemented:
            return res
        else:
            return not res

    def __hash__(self):
        return hash(self.pos)

    def _kCompare(self, other):
        if isinstance(other, self.__class__):
            # Lexicographic sorting
            if other.k[0] == float("inf") and self.k[0] == float("inf"):
                first = 0
            else:
                first = other.k[0] - self.k[0]

            if first == 0:
                if other.k[1] == float("inf") and self.k[1] == float("inf"):
                    second = 0
                else:
                    second = other.k[1] - self.k[1]
                if second == 0:
                    if other.pos > self.pos:
                        return 1
                    else:
                        return -1

            if first != 0:
                return first
            else:
                return second
        else:
            return NotImplemented
```

```
    def __lt__(self, other):
        return self._kCompare(other) > 0

    def __gt__(self, other):
        return self._kCompare(other) < 0

    def __le__(self, other):
        return self._kCompare(other) >= 0

    def __ge__(self, other):
        return self._kCompare(other) <= 0

    def pred(self):
        """Possible parents of this node.

        This should be overridden by child classes."""
        pass

    def succ(self):
        """Possible children of this node.

        This should be overridden by child classes."""
        pass

    def __str__(self):
        return "pos: %s, k: %s" % (self.pos, self.k)


class LPA():
    def __init__(self, sStart, sGoal, state_factory, stateDict, debug=False):
        """Initializes LPA* class.

        Args:
            sStart: Starting state (type: state)
            sGoal: Ending state (type: state)
            stateDict: A dictionary to find the most current version of a state.
        """

        # U is a priority queue of inconsistent nodes (g != rhs)
        self._U = pq.Queue()

        # Warning that for these two dicts, the keys compare by position,
        # but the key isn't updated with the latest k value
        # This shouldn't cause any problems in any reasonable use.

        # Basically, just don't ever use the keys for anything except insert/retrieve

        self._g_dict = {}
        self._rhs_dict = {}

        self._impassable_nodes = set([])
        self._impassable_edges = set([])

        self.sGoal = sGoal
        self.sStart = sStart
        self.debug = debug
        self.state_factory = state_factory

        self._rhs_dict[sStart] = 0

        sStart.k = self._calculateKey(sStart)
        self._U.insert(sStart)

        self.stateDict = stateDict
        self.stateDict[sStart] = sStart

    def _g(self, s):
```

```
        return self._g_dict.get(s, float("inf"))

    def _rhs(self, s):
        if s == self.sStart:
            return 0
        else:
            return self._rhs_dict.get(s, float("inf"))

    def _printGRHS(self):
        print "The state of g, rhs:"
        candidates = set(self._g_dict.keys())
        candidates.update(set(self._rhs_dict.keys()))
        for k in candidates:
            print "%s: (%s, %s)" % (k.pos, self._g(k), self._rhs(k))

    def _printQueue(self):
        print "the state of the queue:"
        self._U.printQueue()

    def _printRHS(self):
        print "The state of rhs:"
        for k, v in self._rhs_dict.iteritems():
            print "%s: %s" % (k.pos, self._rhs(k))


    def _calculateKey(self, s):
        """Updates the key for a specific ndoe."""
        return (min(self._g(s), self._rhs(s)) + self._h(s, self.sGoal),
                min(self._g(s), self._rhs(s)))

    def _h(self, s, sGoal):
        """Heuristic obeying the triangle inequality. This should be overridden in
    derivative classes."""
        pass

    def _c(self, s, s2):
        """Cost to move from s to s2. This should be overridden in derivative classes
    ."""
        pass

    def _updateVertex(self, u):
        if u != self.sStart:
            # Update the estimate with lowest cost from predecessors
            preds = [self.state_factory.makeOrGetStateByPos(pos) for pos in u.pred()]
            self._rhs_dict[u] = min([self._g(s) + self._c(s, u) for s in preds])
        if u in self._U:
            self._U.remove(u)

        if self._g(u) != self._rhs(u):
            u.k = self._calculateKey(u)
            self._U.insert(u)
            self.stateDict[u] = u
            self.state_factory.updateState(u)


    def computeShortestPath(self):
        while (self._U.topKey() < self._calculateKey(self.sGoal) or
               self._rhs(self.sGoal) != self._g(self.sGoal)):
            if self.debug: self._printQueue()
            u = self._U.pop()
            if self.debug: print "now looking at %s" % (u.pos,)
            if self.debug: self._printGRHS()
            if self._g(u) > self._rhs(u):
                self._g_dict[u] = self._rhs(u)
                for pos in u.succ():
                    s = self.state_factory.makeOrGetStateByPos(pos)
                    if self.debug: print "updating %s" % (s.pos,)
                    self._updateVertex(s)
```

```
                if self.debug: self._printGRHS()
            else:
                self._g_dict[u] = float("inf")
                for pos in u.succ():
                    s = self.state_factory.makeOrGetStateByPos(pos)
                    self._updateVertex(s)

    def getShortestPath(self):
        # Make sure to only run this after computeShortestPath
        if self.debug: print "\tgetting shortest path"
        sCur = self.sGoal
        path = [sCur]
        totalCost = 0

        while sCur != self.sStart:
            if self.debug: print "\tfinding new node to add"
            minCost = float("inf")
            sNext = None
            for sPredPos in sCur.pred():
                sPred = self.state_factory.makeOrGetStateByPos(sPredPos)
                if sPred in path:
                    continue
                tmpCost = self._g(sPred) + self._c(sPred, sCur)
                if self.debug: print("\tgot a min cost of %s with a tmp cost of %s"
                    % (minCost, tmpCost))
                minCost = min(minCost, tmpCost)
                if tmpCost == minCost and tmpCost != float("inf"):
                    moveCost = self._c(sPred, sCur)
                    sNext = sPred

            if sNext is None:
                return (None, float("inf"))
            sCur = sNext
            path.append(sCur)
            if len(path) > 20:
                for item in path:
                    print item.pos
                raise PathTooLongException
            totalCost += moveCost

        if self.debug: print totalCost
        return (path[::-1], totalCost)

    def makeNodeImpassable(self, pos):
        """This should be overridden in derivative classes."""
        pass

    def makeEdgeImpassable(self, edge):
        """This should be overridden in derivative classes."""
        pass

    def getConstraints(self):
        return (self._impassable_nodes, self._impassable_edges)

    def printConstraints(self):
        print("Impassable nodes:")
        for node in self._impassable_nodes:
            print "\t%s" % (node.pos,)

        print("\nImpassable edges:")
        for edge in self._impassable_edges:
            print "\t%s—%s" % (edge[0], edge[1])
```

A.4 lpa_math.py

```
"""Handy math functions for fluid routing."""
```

```
import numpy as np

def closestDistanceBetweenLines(a0, a1, b0, b1, clampAll=True,
                                clampA0=False, clampA1=False,
                                clampB0=False, clampB1=False):

    ''' Given two lines defined by numpy.array pairs (a0,a1,b0,b1)
        Return the shortest distance between them
    '''

    # From
    # http://stackoverflow.com/questions/2824478/shortest-distance-between-two-line-
    segments

    # If clampAll=True, set all clamps to True
    if clampAll:
        clampA0 = True
        clampA1 = True
        clampB0 = True
        clampB1 = True


    # Calculate denomitator
    A = a1 - a0
    B = b1 - b0
    magA = np.linalg.norm(A)
    magB = np.linalg.norm(B)

    _A = A / magA
    _B = B / magB

    cross = np.cross(_A, _B)
    denom = np.linalg.norm(cross)**2


    # If lines are parallel (denom=0) test if lines overlap.
    # If they don't overlap then there is a closest point solution.
    # If they do overlap, there are infinite closest positions, but there is a
    closest distance
    if not denom:
        d0 = np.dot(_A, (b0 - a0))

        # Overlap only possible with clamping
        if clampA0 or clampA1 or clampB0 or clampB1:
            d1 = np.dot(_A, (b1 - a0))

            # Is segment B before A?
            if d0 <= 0 >= d1:
                if clampA0 and clampB1:
                    if np.abs(d0) < np.abs(d1):
                        return np.linalg.norm(a0 - b0)
                    return np.linalg.norm(a0 - b1)


            # Is segment B after A?
            elif d0 >= magA <= d1:
                if clampA1 and clampB0:
                    if np.abs(d0) < np.abs(d1):
                        return np.linalg.norm(a1 - b0)
                    return np.linalg.norm(a1 - b1)


        # Segments overlap, return distance between parallel segments
        return np.linalg.norm(((d0*_A) + a0) - b0)
```

```
    # Lines criss-cross: Calculate the projected closest points
    t = (b0 - a0)
    detA = np.linalg.det([t, _B, cross])
    detB = np.linalg.det([t, _A, cross])

    t0 = detA/denom
    t1 = detB/denom

    pA = a0 + (_A * t0) # Projected closest point on segment A
    pB = b0 + (_B * t1) # Projected closest point on segment B


    # Clamp projections
    if clampA0 or clampA1 or clampB0 or clampB1:
        if clampA0 and t0 < 0:
            pA = a0
        elif clampA1 and t0 > magA:
            pA = a1

        if clampB0 and t1 < 0:
            pB = b0
        elif clampB1 and t1 > magB:
            pB = b1

        # Clamp projection A
        if (clampA0 and t0 < 0) or (clampA1 and t0 > magA):
            dot = np.dot(_B, (pA - b0))
            if clampB0 and dot < 0:
                dot = 0
            elif clampB1 and dot > magB:
                dot = magB
            pB = b0 + (_B * dot)

        # Clamp projection B
        if (clampB0 and t1 < 0) or (clampB1 and t1 > magB):
            dot = np.dot(_A, (pB - a0))
            if clampA0 and dot < 0:
                dot = 0
            elif clampA1 and dot > magA:
                dot = magA
            pA = a0 + (_A * dot)


    return np.linalg.norm(pA - pB)
```

A.5 Priority Queue Implementations

A.5.1 priority_queue_binheap.py

```
"""Implements a priority queue for LPA*"""

import heapq


class Queue():
    """Binomial heap augmented with hashtable.

    Note: This can be copy.deepcopy'd correctly."""
    def __init__(self):
        self._U = []
        self._set = set([])

    def insert(self, item):
        heapq.heappush(self._U, item)
        self._set.add(item)
```

```
    def top(self):
        if len(self._U) > 0:
            return self._U[0]
        else:
            return None

    def topKey(self):
        t = self.top()
        if type(t) == tuple:
            return t
        else:
            if t is None:
                return (float("inf"), float("inf"))
            else:
                return t.k

    def pop(self):
        item = heapq.heappop(self._U)
        self._set.remove(item)
        return item

    def remove(self, item):
        self._set.remove(item)
        self._U.remove(item)
        heapq.heapify(self._U)


    def __contains__(self, key):
        return key in self._set

    def __len__(self):
        return len(self._U)

    def printQueue(self):
        for u in self._U:
            print u

    def __str__(self):
        toReturn = "note: this destroys the queue\n\n"
        while len(self._U) > 0:
            u = self.pop()
            toReturn += "%s\n" % (str(u),)
        return toReturn
```

A.5.2 priority_queue_linked_list.py

```
"""Implements a priority queue for LPA*"""


class Queue():
    """Linked list augmented with hashtable.

    Note: This can be copy.deepcopy'd correctly."""
    def __init__(self):
        self._U = {}
        self._list_U = None

    def insert(self, item):
        listItem = LinkedList(item)
        if self._list_U == None:
            self._list_U = listItem
        else:
            self._list_U.add(listItem)
            if self._list_U.prev:
                self._list_U = self._list_U.prev
        self._U[item] = listItem
```

```
    def top(self):
        if self._list_U:
            return self._list_U.val
        else:
            return None

    def topKey(self):
        t = self.top()
        if type(t) == tuple:
            return t
        else:
            if t is None:
                return (float("inf"), float("inf"))
            else:
                return t.k

    def pop(self):
        topListItem = self._list_U
        self._list_U = topListItem.next
        item = topListItem.pop()
        del self._U[item]
        return item

    def remove(self, item):
        listItem = self._U[item]
        if listItem == self._list_U:
            self._list_U = self._list_U.next
        listItem.pop()
        del self._U[item]


    def __contains__(self, key):
        return key in self._U

    def __len__(self):
        return len(self._U)

    def printQueue(self):
        for u in self._U:
            print u

class LinkedList():
    def __init__(self, val):
        self.val = val
        self.next = None
        self.prev = None

    def pop(self):
        if self.prev:
            self.prev.next = self.next
        if self.next:
            self.next.prev = self.prev
        return self.val

    def add(self, other):
        if self.val > other.val:
            other.prev = self.prev
            other.next = self
            if self.prev:
                self.prev.next = other
            self.prev = other
        else:
            if self.next:
                self.next.add(other)
            else:
                other.prev = self
                self.next = other
```

```
    def __str__(self):
        if self.prev and self.next:
            selfStr = "(%s <- %s -> %s)" % (self.prev.val.k, self.val.k, self.next.
    val.k)
        elif self.prev:
            selfStr = "(%s <- %s -X)" % (self.prev.val.k, self.val.k)
        elif self.next:
            selfStr = "(X- %s -> %s)" % (self.val.k, self.next.val.k)
        else:
            selfStr = "(X- %s -X)" % (self.val.k,)
        return selfStr + ", " + str(self.next)
```

A.5.3 priority_queue_array.py

```
"""Implements a priority queue for LPA*"""


class Queue():
    """Array augmented with hashtable.

    Note: This can be copy.deepcopy'd correctly."""
    def __init__(self):
        self._U = []
        self._set = set([])

    def insert(self, item):
        self._U.append(item)
        self._set.add(item)
        self._U.sort()

    def top(self):
        if len(self._U) > 0:
            return self._U[0]
        else:
            return None

    def topKey(self):
        t = self.top()
        if type(t) == tuple:
            return t
        else:
            if t is None:
                return (float("inf"), float("inf"))
            else:
                return t.k

    def pop(self):
        item = self._U.pop(0)
        self._U.sort()
        self._set.remove(item)
        return item

    def remove(self, item):
        self._set.remove(item)
        self._U.remove(item)


    def __contains__(self, key):
        return key in self._set

    def __len__(self):
        return len(self._U)

    def printQueue(self):
        for u in self._U:
            print u
```

A.5.4 priority_queue_treap.py

```
"""Implements a priority queue for LPA*"""

import treap


class Queue():
    """Treap augmented with hashtable.

    Note: This can be copy.copy'd correctly.
    It mostly works, but Cython extensions get unhappy."""
    def __init__(self):
        self._treap = treap.treap()

    def insert(self, item):
        self._counter += 1
        self._treap[item] = 1

    def top(self):
        try:
            return self._treap.find_min()
        except KeyError:
            return None

    def topKey(self):
        t = self.top()
        if type(t) == tuple:
            return t
        else:
            if t is None:
                return (float("inf"), float("inf"))
            else:
                return t.k

    def pop(self):
        try:
            item = self._treap.find_min()
            self._treap.remove_min()
        except KeyError:
            item = None
        return item

    def remove(self, item):
        self._treap.remove(item)

    def __contains__(self, key):
        return key in self._treap

    def __len__(self):
        return len(self._treap)

    def printQueue(self):
        print(self._treap)

    def __str__(self):
        toReturn = "note: this destroys the queue\n\n"
        while len(self._treap) > 0:
            u = self.pop()
            toReturn += "%s\n" % (str(u),)
        return toReturn
```

A.5.5 priority_queue_sortedset.py

```
"""Implements a priority queue for LPA*"""
```

```
import sortedcontainers
import copy

class Queue():
    """Nested Lazy-Length Lists augmented with hashtable.

    Note: This can be copy.deepcopy'd correctly."""
    def __init__(self):
        self._set = sortedcontainers.SortedSet()

    def insert(self, item):
        self._set.add(item)

    def top(self):
        try:
            return self._set[0]
        except IndexError:
            return None

    def topKey(self):
        t = self.top()
        if type(t) == tuple:
            return t
        else:
            if t is None:
                return (float("inf"), float("inf"))
            else:
                return t.k

    def pop(self):
        try:
            item = self._set.pop(0)
        except IndexError:
            item = None
        return item

    def remove(self, item):
        self._set.remove(item)

    def __contains__(self, key):
        return key in self._set

    def __len__(self):
        return len(self._set)

    def printQueue(self):
        print(self._set)


    def __deepcopy__(self, memo):
        result = Queue()
        memo[id(self)] = result
        tmpItem = self.top()
        for item in self._set:
            result.insert(copy.deepcopy(item, memo))
        return result
```

Appendix B

Profiling Results

B.1 Binomial Heap

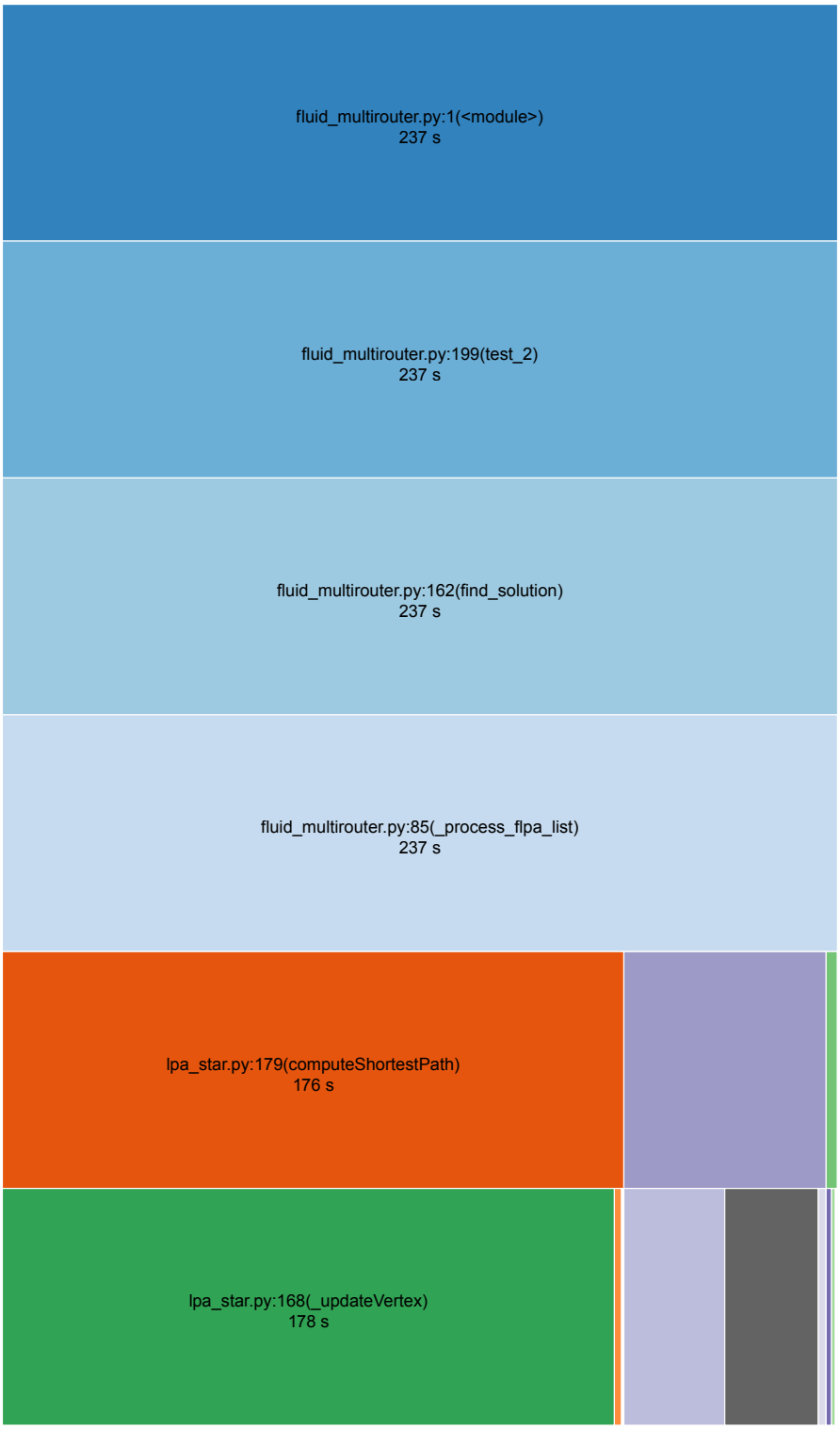


Figure B-1: Binomial Heap priority queue profiling results.

B.2 Linked List

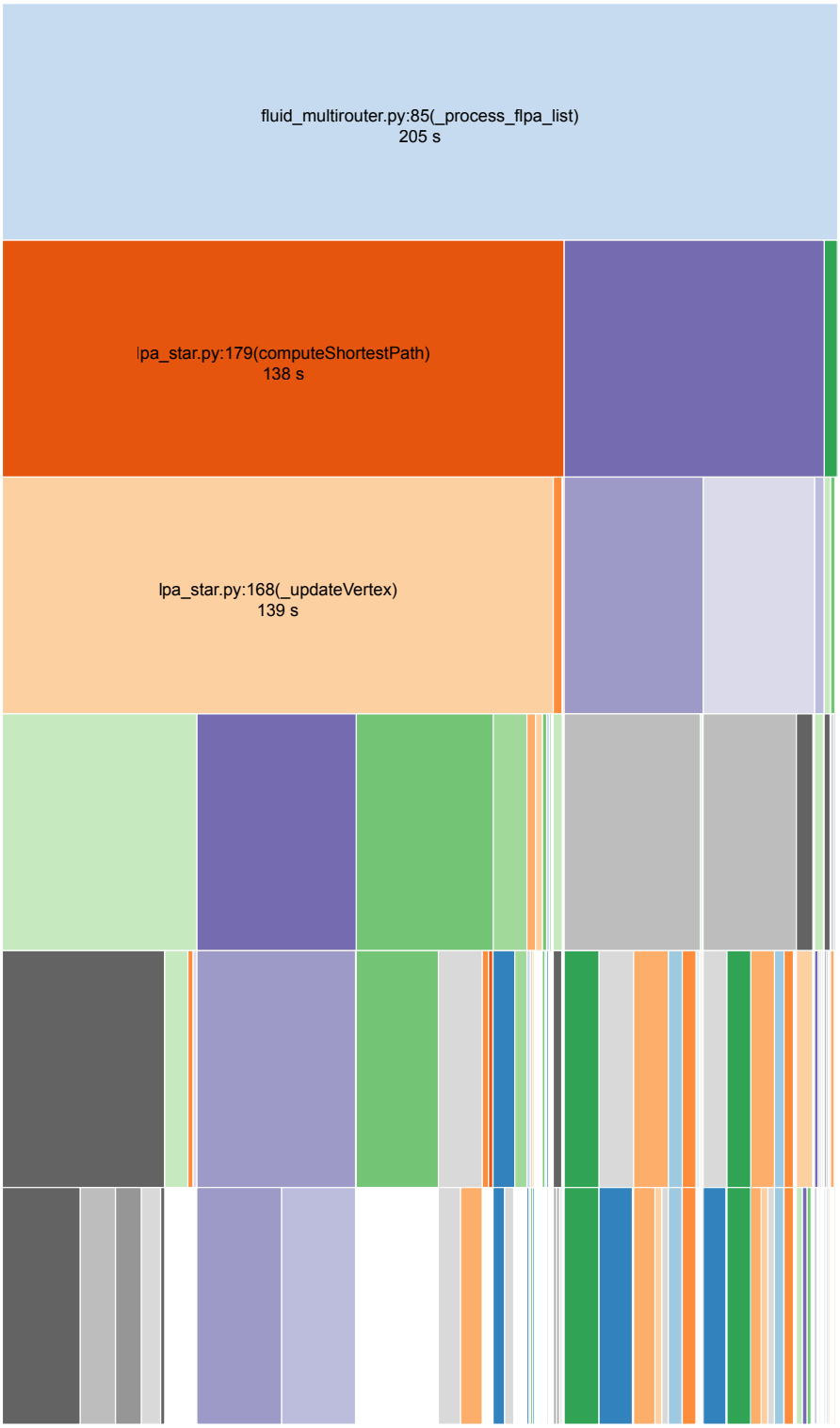


Figure B-2: Linked List priority queue profiling results.

B.3 Array

Figure B-3: Array priority queue profiling results.

B.4 Treap

Figure B-4: Treap priority queue profiling results.

B.5 Nested Lazy-Length Lists

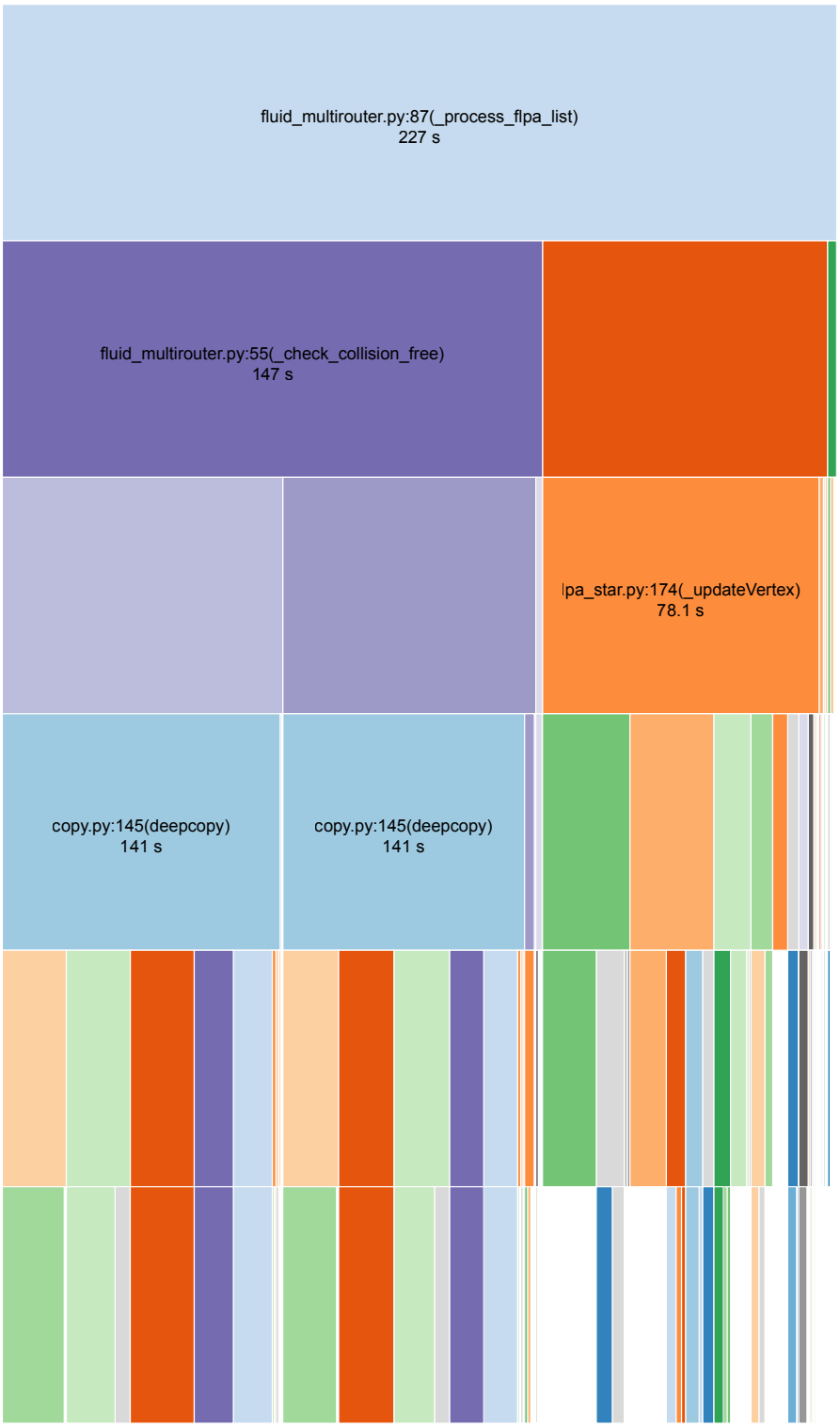


Figure B-5: Nested Lazy-Length Lists priority queue profiling results.

B.6 Nested Lazy-Length Lists with cache

Figure B-6: Nested Lazy-Length Lists priority queue with cache profiling results.

Bibliography

[1] Maxim Likhachev Sven Koenig and David Furcy. Lifelong planning a*. *Artificial Intelligence*, 155(1):93 – 146, 2004.

[2] Robert S Fisher. A multi-pass, multi-algorithm approach to pcb routing. In *Papers on Twenty-five years of electronic design automation*, pages 172–181. ACM, 1988.

[3] Guy G Lemieux and Stephen D Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. In *Proc. ACM/SIGDA Physical Design Workshop, Lake Arrowhead, CA*, pages 215–226, 1993.

[4] Ronald L Rivest and Charles M Fiduccia. A âĂIJgreedyâĂİ channel router. In *Proceedings of the 19th Design automation conference*, pages 418–424. IEEE Press, 1982.

[5] Michael Burstein and Richard Pelavin. Hierarchical channel router. In *Proceedings of the 20th Design Automation Conference*, pages 591–597. IEEE Press, 1983.

[6] Sabih H Gerez and Otto E Herrmann. Packer: a switchbox router based on conflict elimination by local transformations. In *Circuits and Systems, 1989., IEEE International Symposium on*, pages 961–964. IEEE, 1989.

[7] David N Deutsch. A âĂIJdoglegâĂİ channel router. In *Proceedings of the 13th Design Automation Conference*, pages 425–433. ACM, 1976.

[8] KC Chang and DH-C Du. Efficient algorithms for layer assignment problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(1):67–78, 1987.

[9] Thomas G Szymanski. Dogleg channel routing is np-complete. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4(1):31–41, 1985.

[10] Christopher Condrat, Priyank Kalla, and Steve Blair. Crossing-aware channel routing for photonic waveguides. In *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, pages 649–652. IEEE, 2013.

[11] R. MacCurdy, R. Katzschmann, Youbin Kim, and D. Rus. Printable hydraulics: A method for fabricating robots by 3d co-printing solids and liquids. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3878–3885, May 2016.

[12] Thiago Pereira, Szymon Rusinkiewicz, and Wojciech Matusik. Computational light routing: 3D printed fiber optics for sensing and display. *ACM Transactions on Graphics*, 33(3), May 2014.

[13] Valkyrie Savage, Ryan Schmidt, Tovi Grossman, George Fitzmaurice, and Björn Hartmann. A series of tubes: Adding interactivity to 3d prints using internal pipes. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pages 3–12, 2014.

[14] Daniel Stromberg. Treap: Python implementation of treaps, 2009. [Online: https://pypi.org/project/treap/; accessed 5/25/18].

[15] Grant Jenks. Sortedcontainers: Sorted list, sorted dict, sorted set, 2014. [Online: https://pypi.org/project/sortedcontainers/; accessed 5/25/18].

[16] Stelian Coros, Bernhard Thomaszewski, Gioacchino Noris, Shinjiro Sueda, Moira Forberg, Robert W Sumner, Wojciech Matusik, and Bernd Bickel. Computational design of mechanical characters. *ACM Transactions on Graphics (TOG)*, 32(4):83, 2013.

[17] Bernhard Thomaszewski, Stelian Coros, Damien Gauge, Vittorio Megaro, Eitan Grinspun, and Markus Gross. Computational design of linkage-based characters. *ACM Transactions on Graphics (TOG)*, 33(4):64, 2014.

[18] Gaurav Bharaj, Stelian Coros, Bernhard Thomaszewski, James Tompkin, Bernd Bickel, and Hanspeter Pfister. Computational design of walking automata. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, SCA '15, pages 93–100, New York, NY, USA, 2015. ACM.

[19] Adriana Schulz, Cynthia Sung, Andrew Spielberg, Wei Zhao, Yu Cheng, Ankur Mehta, Eitan Grinspun, Daniela Rus, and Wojciech Matusik. Interactive robogami: Data-driven design for 3d print and fold robots with ground locomotion. In *SIGGRAPH 2015: Studio*, SIGGRAPH '15, pages 1:1–1:1, New York, NY, USA, 2015. ACM.

[20] Cynthia Sung and Daniela Rus. Foldable joints for foldable robots. *Journal of Mechanisms and Robotics*, 7(2):021012, 2015.

[21] Mark Fuge, Greg Carmean, Jessica Cornelius, and Ryan Elder. The mechprocessor: Helping novices design printable mechanisms across different printers. *Journal of Mechanical Design*, 137(11):111415, 2015.

[22] Mark M Plecnik and J Michael McCarthy. Computational design of stephenson ii six-bar function generators for 11 accuracy points. *Journal of Mechanisms and Robotics*, 8(1):011017, 2016.

[23] JA Cabrera, A Simon, and M Prado. Optimal synthesis of mechanisms with genetic algorithms. *Mechanism and machine theory*, 37(10):1165–1177, 2002.

[24] Damir Vucina and Ferdinand Freudenstein. An application of graph theory and nonlinear programming to the kinematic synthesis of mechanisms. *Mechanism and machine theory*, 26(6):553–563, 1991.

[25] VN Sohoni and EJ Haug. A state space technique for optimal design of mechanisms. *Journal of Mechanical Design*, 104(4):792–798, 1982.

[26] Alexander Pasko, Valery Adzhiev, Alexei Sourin, and Vladimir Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.

[27] AAG Requicha. Mathematical models of rigid solids: Theory, methods and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.

[28] Python Software Foundation. Python language reference, version 2.7. [Online: https://docs.python.org/2/library/profile.html; accessed 5/25/18].

[29] Matt Davis. Snakeviz: A web-based viewer for python profiler output, 2012. [Online: https://pypi.org/project/snakeviz/; accessed 5/25/18].